1998 International Computer Symposium
Workshop on Image Processing and Character Recognition
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

VECTORIZATION USING BRESENHAM LINES

Lijung Jiang and Jiang-Hsing Chu

Department of Computer Science
Southern Illinois University at Carbondale
Carbondale, IL 62901, USA
jchu@cs.siu.edu

ABSTRACT

We study vectorization algorithms which vectorize raster into vector files consisting of lines with the restriction that pixels corresponding to a line can be fully recovered by applying the Bresenham line drawing algorithm on the endpoints of the line. We also study the problem of finding a minimum number of line segments which is needed to represent a given raster. We prove that this problem is NP-complete and present an efficient approximation algorithm which produces near optimal results.

1. INTRODUCTION

Computer images are stored mainly in two types of formats: vector and raster. A vector image contains descriptions of picture objects such as lines and circles that make up the image. For example, a vector file may contain the image information like "line (10,20)(30,30), line (30,30)(40,50), line (40,50)(100,50)..." A vector output device such as a plotter interprets a vector file and draw the corresponding image. On the other hand, a raster image consists of a two-dimensional array of tiny dots that could be screen pixels or printer dots. We will simply call these dots pixels hereafter. A raster output device displays a raster image by setting the corresponding pixels on. Pixels which are on will be referred to as black pixels hereafter. In a raster file of a binary image, each pixel is represented by a bit. More bits per pixel are needed in images with more gray levels or color images. As a result, the raster files are usually considerably larger than the vector files.

Almost all computers today are equipped with raster output devices such as monitors and laser printers. These devices can display raster images as well as vector images, thank to the development of the rasterization algorithms that produce, on the raster devices, the image described by vector files. A good example is the famous Bresenham [1] line drawing algorithm which draws straight lines on raster devices. Algorithms for drawing circles and ellipses are also available [4].

Eastman [2] indicated that current engineering drawing systems are based on vector display technology because vector system provides many benefits during revision. For example, it is easy to delete a portion of a dense drawing perfectly. Also, the vector system has minimum loss of precision in rotation or scaling. Although raster images can display solid shaded images very well, they are hard to rotate and scale due to the absence of underlying logic structures in most raster systems. In general, the raster systems are best for shaded images and the vector systems are best for engineering line drawing. Therefore, most CAD systems are use the vector system. This causes a problem to many engineering companies who have archives of designs or blueprints and would like to use a CAD system to edit them.

As a summary, there are two main advantages for converting the raster images into vector images:

Compression: unless the image is very complicated, a vector file is usually much smaller than its corresponding raster file. As a result, a vector file uses less memory, less disk space, and less transferring time.

Compatibility: since some applications and some devices only deal with the vector images, we must covert the raster image into vector form. For example, if we are going to edit a raster image using a vector editor such as a CAD editor, the image needs to be converted.

Parker [5] developed a vecotrization algorithm which uses the chord property to find the set of pixels that are covered by a given line. However, the method is not compatible with the Bresenham line drawing algorithm. To use vectorization algorithm as an image compression tool, we need to guarantee that the vector file obtained through vectorization can be converted back to the original raster image. Because many display devices use Bresenham line drawing algorithm to draw vector lines on raster devices, so it is important to make vectorization algorithm compatible with Bresenham line drawing algorithm.

In this paper, we present a vectorization method which is based on the Bresenham line drawing algorithm. But first we study the problem of finding a minimum number of lines to cover all black pixels in a raster.

1998 International Computer Symposium
Workshop on Image Processing and Character Recognition
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

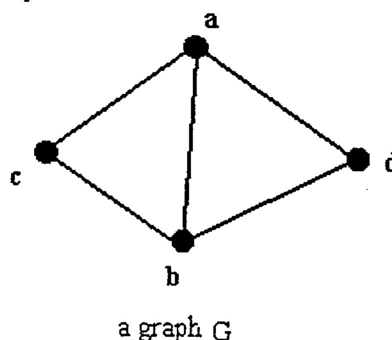
2. NP-COMPLETENESS

The problem we are trying to solve is: Can we find a minimum number of Bresenham lines to cover all black pixels in a raster? Whenever we are confronted with a new problem without an obvious polynomial time solution, a question arises naturally: Is this an NP-complete problem?
Let us first analyze this problem. In order to find a minimum number of lines, we can first find all possible *useful* lines. We say a line is *useless* if all its pixels are covered by another line. We use L_{all} to represent the set of all useful lines. We can then divide the set L_{all} into two subsets L_s and L_d. L_s contains only the lines that cover at least one pixel that is not covered by any other lines in L_{all}. The rest of the lines in L_{all} are belong to L_d ($L_d = L_{all} - L_s$). We use P to represent the black pixels. Similarly we divide P into two sets P_s and P_d. The set P_s contains all pixels which are covered by at least one line in L_s. The set P_d contains the rest of the pixels in P which are not covered by any line in L_s ($P_d = P - P_s$). Obviously any set with a minimum number of lines that covers all pixels in P must contain all lines in L_s. So what is left is to find a minimum number of lines from L_d to cover all pixels in P_d. It is worth mentioning that lines in L_d may also cover some pixels covered by P_s, but we need not worry about these pixels since they are covered by lines in L_s, which are always included in the final set. Now our problem can be reduced to the following decision problem :

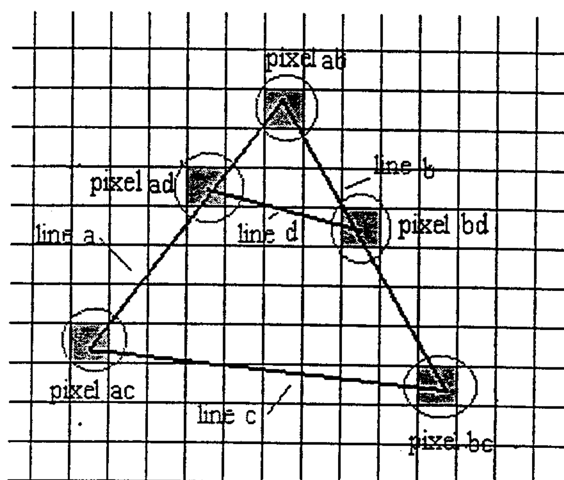
> **INSTANCE:** An image R=(L,P), where L is a set of lines and P is a set of pixels with every pixel in P being covered by at least two lines in L, and a positive integer K.
>
> **QUESTION:** Is there a subset $L' \subseteq L$ such that $|L'| \leq K$ and, every pixel in P is covered by at least one line in L'. In other words, is there a subset of L with K or less lines which cover all pixels in P.

We will refer to this problem as the **LINE COVER (LC)** problem. We now prove that **LC** is NP-complete. To prove that a problem is NP-complete, we need to show that the problem is in NP and also reduce an NP-complete problem to the problem in polynomial time [3]. It is easy to see that **LC** is in NP since a non-deterministic algorithm needs only guess a subset of L and check in polynomial time to see whether every pixel in P is covered by at least one line in this subset. To prove that it is NP-complete, we reduce the well known [**VERTEX COVER (VC)** problem to **LC**.

Now we describe a way to transform any instance of **VC**, a graph G=(V, E) and a positive integer $K \leq |V|$, to a raster image R (R with a pixel set P and lines as an instance of **LC**), so that R has a line cover of size K or less if and only if G has a vertex cover of size K or less. The transformation is straightforward. The main idea is to transform a vertex to a line in R, and an edge to a pixel in R. To be more precise, an edge is transformed to a pixel which is covered by the two lines corresponding to the two vertices incident to the edge. An example is shown in Figure 2.1. In R, line a corresponds to vertex **a** in G, line c to **c**, line b to **b**, line d to **d**. pixel ac to edge **ac**, pixel ab to edge **ab**, pixel bc to edge **bc**, pixel ad to edge **ad**, pixel bd to **bd**



a graph G



a corresponding transformation in R

Figure 2.1 a transformation of a graph G

A raster that is created by the transformation must meet the following conditions:

1. Different vertices in G correspond to different lines in R, and different edges in G correspond to different pixels in R
2. Every pixel in R is covered by exactly two lines.
3. If (u,v) is an edge in G, then the pixel it transforms to must be a member of the intersection of the two lines l_u and l_v. If (u,v) is not an edge, then none of the pixels in the intersection of l_u and l_v should be included in the pixel set P.

It is easy to see how the construction can be accomplished in polynomial time. The graph R we get through this transformation will contain |V|. We now show that the G has a vertex cover of size K or less if and only if R has a line cover of size K or less.

First, suppose that there is a line cover L' in R which covers all pixels and $|L'| \leq K$. In Figure 2.1, if line a in R is selected, it covers pixel ac, pixel ab, and pixel ad. Because of the conditions about the raster stated earlier, if we select the vertices corresponding to the lines in L' as our element of V', V' will cover all edges in G and $|V'| \leq K$.

Conversely, suppose that $V' \subseteq V$ is a vertex cover for G with $|V'| \leq K$. For each vertex in V', it covers all edges which are incident to this vertex. Its corresponding line in R covers all pixels which correspond to those edges in G. So if we select all lines in R which correspond to the vertices in V' to form a line cover L', then $|L'| \leq K$, and L' will cover all pixels in R.

We have proved that the **LC** problem is NP-complete. Our original problem is a problem in P (finding L_{all} and extracting L_s from L_{all} can be finished in polynomial time) combined with a NP-complete problem (**LC**), which is NP-complete.

3. THE OPTIMAL ALGORITHM

Now that we have proved that the problem we want to solve is NP-complete, it is not likely that we can construct an efficient algorithm running in polynomial time and produces an optimal solution. We will thus concentrate our efforts in finding an efficient approximation algorithm. Nevertheless it will be useful to have an optimal algorithm so that it can be used as a benchmark to see how well the approximation algorithm performs. In this section we present an optimal algorithm for extracting a minimum number of Bresenham lines in a raster image.

Before we discuss the optimal algorithm, we have the following definitions:

Bresenham Property: A line set is said to have Bresenham Property if and only if the lines in this set will give exactly the same pixels as the original raster image, when we use Bresenham line drawing algorithm to draw them on the raster device.

Bresenham Cover Property: A line set is said to have Bresenham Cover Property if drawn by using Bresenham line drawing algorithm, it covers all black pixels in the raster image.

Static Line: In a Bresenham line set$\{l_1, l_2, l_3, ...l_n\}$, a line l_i is static if it covers at least one pixel that is not belong to any other line $l_j(j \neq i)$. Otherwise we say the line is dynamic.

Useless Line: A line is a useless line, if all its pixels are covered by another line.

As described in section 3, we first try to obtain all possible lines L_{all}, then find the static lines set L_s from these lines. Recall that $L_d = L_{all} - L_s$. Then we can divide all the pixels in P into P_s and P_d. All we need to do now is finding a line set L' with a minimum number of lines from L_d which cover all pixels in P_d. Put L' and L_s together, we will get the result we wanted.

If we try to get all possible lines L_{all} at once, potentially there will have a lot Bresenham lines that can be generated. It will need a lot of memory to store these possible lines. To avoid this situation, our approach is getting all possible lines gradually. In each step, we divide the L_{all} into L_1 and L_2. L_1 is used to store static lines and some dynamic lines so far. L_2 used to store the dynamic lines (i.e., all pixels of the line are covered by some lines in L_1, but they are not useless lines, they may be needed later to find the set with minimum number of lines), the useless lines are discarded. After we obtain all possible lines, the L_1 has **Bresenham Cover Property**. we use N to represent the number of lines in L_1.

We can then extract L_s from L_1, which is straightforward. Let us use N_s to represent the number of lines in L_s. The only thing needs to be done is to find a subset L' with minimum number of lines from L_d, $L_d = L_2 + L_1 - L_s$. In order to reduce the number of lines in L_d, We discard the lines in L_d whose pixels can be covered by lines in L_s. We will erase all pixels which can be covered by lines of L_s. We know that if the minimum number of lines is N', it means we can get at least one line set whose line number is N', and this set has **Bresenham Cover Property**. Because the N' is the minimum possible value, so we can not get any line set of less than N' lines and satisfies the **Bresenham Cover Property**. In order to reduce computing time, we first try all possible line sets whose lines' number is $(N - N_s - 1)$ in L_d. We will check all such sets to see whether the **Bresenham Cover Property** is satisfied in any of them. If none of these sets has the **Bresenham Cover Property**, we will stop our algorithm, and we can conclude that the N is the minimum value, the line set L_1 is the optimal line set we can get from P_d. Otherwise, we will store the set who has **Bresenham Cover Property** into L_1, then we will repeat search for a line set with one fewer lines, until we can not find such line set, finally, the minimum lines set will be L_s plus L_1.

Below we list our algorithm in a C-like syntax. In our algo-

rithm, Assume the raster image in a screen buffer (two dimensional array). We define the lower left corner of screen to be (0, 0), the lower right corner to be (0, WIDTH), the upper left corner to be (0, HEIGHT), and the upper right corner to be (WIDTH, HEIGHT).

Algorithm: extracting minimum Bresenham lines in the input raster image
Input : Raster image with WIDTH * HEIGHT resolution
Output: A vector file contain minimum number of Bresenham lines

```
/* get the all possible lines */
L1 =∅ ;    /* initialize the L1 to empty */
L2 =∅;     /* initialize the L2 to empty */
for( x1= 0; x1 <WIDTH; x1 ++)
  for( y1= 0; y1 < HEIGHT; y1 ++) {
    if (Screen[x1, y1]= 1 ){ /* 1 means the pixel(x1, y1 ) is on */
      for( x2 =WIDTH-1; x2 >= x1; x2 --)
        for( y2 = y1; y2 <= HEIGHT; y2++) {
/* using Bresenham line drawing alg. to draw the line from(x1, y1) ,
(x2, y2). if all pixels of this line is turned on in the raster image(i.e.
Screen[x,y]=1) then return true, otherwise return false */
          if( Is_a_Bresenham_Line(x1 , y1, x2, y2)){
            if(Is_Usefull_Line(L1,N,x1,y1,x2,y2)&&
              Is_Usefull_Line(L2,N2, x1, y1,x2, y2)){
              Add_to_L1(L1, N, x1, y1, x2, y2); /* N is #r of lines in L1 */
              /* move some redundant dynamic lines from L1 to L2 */
              Migrate_Line(L1, N, L2, N2);  /* N2 is # of line in L2 */
            }
          }
        }
      }
  }
/* get the minimum number of lines set */
Minimize_Line(L2, N2);  /*discarded useless lines in L2 */
Extract_Static_Line(Lstatic, L1, N); /* get Ls and L1 = L1 - Ls */
Merg_Line(Lall, L1, L2, N, N2);    /* Lall=L1 + L2 */
for( N'=N-1; N'>=1; N--){ /* N is the number of lines in L1 */
/* get all possible set Lo with lines' number N', if any set has Bre-
senham Cover Property, the function return TRUE, otherwise re-
turn false. */
  if(Find_ set_ lines(Lall, Lo, N'){
    N=N';
    L1 =Lo;
  }
  else break;
}
Return (L1 + Ls);
```

In the function Migration_Line(L_1, N, L_2, N2), if we find a line whose pixels are included by other lines in L_1, then we move it to the L_2. So there will never be redundant lines in L_1. After search for all possible useful lines is finished, we use the Extract_Static_Line(L_s, L_1, N, L_2, N, N2) function to get the static lines from L_1. That is if any pixel of a line in L_1 is covered only by itself, the line is put in L_s. After getting all static lines, we will discard the lines in L_2 whose all pixels are covered by the lines in L_s.

4. THE APPROXIMATION ALGORITHM

In this section, we present an approximation algorithm which is based on the optimal algorithm. The first different part of our approximation algorithm from optimal algorithm is the function Add_to_L_1(L_1, N, x_1, y_1, x_2, y_2). In approximation algorithm, we will discard the line when we try to add it to L_1 but find its all pixels can be covered by other lines in L_1, we do this because this line is the shorter line, so it has a smaller chance of being need at later. We also move the dynamic lines from L_1 to L_2 using Migrate_Line(L_1, N, L_2, N2) function. After we stop to find the possible lines, L_1 will have **Bresenham Property**. We don't extract L_s from L_1. We use heuristic method to minimize our line number in L_1, we fetch one line from L_2, put it to L_1, then check to see whether or not we can move two or more lines from L_1 and don't change L_1's **Bresenham Property**. If we can, the number of lines in L_1 will reduced. We try all lines in L_2 using this idea, when L_2 is empty, we will stop, and return L_1 as our result.

The algorithm is in following:
Input : Raster image with WIDTH x HEIGHT resolution
Output: A vector file contain minimum number of Bresenham lines

```
/* get the all possible lines */
L1 =∅;
L2 =∅;
for( x1= 0; x1 <WIDTH; x1 ++)
  for( y1= 0; y1 < HEIGHT; y1 ++) {
    if (Screen[x1, y1]= 1 ){
      for( x2 =WIDTH-1; x2 >= x1; x2 --)
      for( y2 = y1; y2 <= HEIGHT; y2++) {
/* using Bresenham line drawing alg. to draw the line from(x1, y1) ,
(x2, y2). if all pixels of this line is turned on in the raster image(i.e.
Screen[x,y]=1) then return true, otherwise return false */
        if( Is_a_Bresenham_Line(x1 , y1, x2, y2)) {
          if(Is_Usefull_Line(L1,N,x1,y1,x2,y2)&&
            Is_Usefull_Line(L2,N2, x1, y1,x2, y2)) {
            Add_to_L1(L1, x1, y1, x2, y2);
            /* move redundant line in L1 to L2 */
            Migrate_Line(L1, N, L2, N2);   /* N2 is # of line in L2 */
          }
        }
      }
    }
  }
/* use heuristic method to get the minimal number of lines set */
Minimize_Line(L2, N2);   /*discarded useless lines in L2 */
for(i=1; i <=N2; i++){
  Fetch_a_line( l, L2);    /* fetch a line from L2 */
  Add_a_line( L1, l);     /* add a line to L1 */
  /* try move two or more lines from L1 */
  Try_remove_two_more_lines(L1 ,N));
} /* end for loop */
Return (L1 );
```

	Optimal Algorithm		Approximation Algorithm	
	Execution Time(ms)	# of lines	Execution Time(ms)	# of lines
Figure 5.1-1	30620	7	220	8
Figure 5.1-2	9601	9	110	10
Figure 5.1-3	330930	8	80	8

Table 5.1 the result of vectorizing images

The function Minimize_Line(L_2,N2) is used to reduce the line number in L_2. Because there maybe is a line we move it from L_1 to L_2, it can cover all pixels of some lines migrated early in L_2, so we use this function to remove some useless line from L_2.

5. Empirical Results

We conducted several experiments to see the performance of our algorithm. We implemented our algorithm using Borland-C++ 5.0 running on a PC with 200 MHZ Pentium-Pro processor in a Window 95 environment. We shut down all other applications so that our program was the only running application. The times were measured from the program started till it ended.

5.1 Optimal Algorithm vs. Approximation Algorithm

Our first experiment was to show the performance of our algorithm compared to that of the optimal algorithm. The performance is measured by the number of lines produced by the algorithms to represent the given raster images. Due to the extremely slow running time of the optimal algorithm, we limited the size of input raster images to 16 × 16 with 50 or fewer useful lines. Three raster images, one dense image as shown in Figure 5.1-1, and two sparse images as shown in Figure 5.1-2 and Figure 5.1-3, were used as the input data and the results are listed in Table 5.1 shown below. The images have been magnified so each pixel is shown as a square.

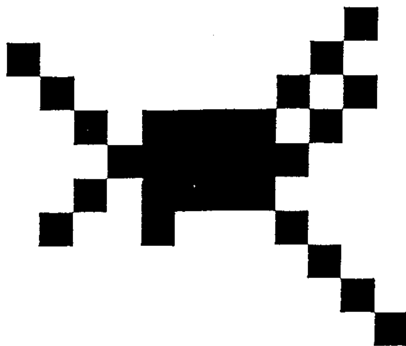


Figure 5.1-1 a dense image

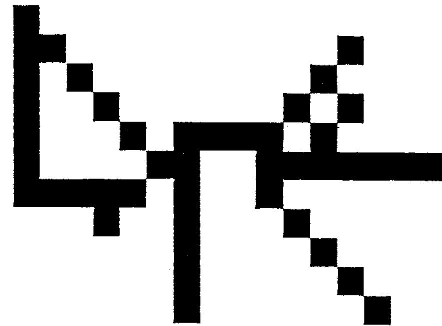


Figure 5.1-2 a sparse image

Figure 5.1-3 another sparse image

The optimal algorithm ran a lot slowly, which is not surprising at all. The difference in execution times is more significant when the input image is dense. The execution times using optimal algorithm depend not only on the density of the image but also on the number of lines in L_d. Figure 5.1-3 has more lines in L_d than Figure 5.1-1 does, and that is the reason why the execution time needed for Figure 5.1-3 is much longer than that of other images. The number of lines produced by the approximation algorithm is quite close to the optimal number in all three cases. It is hard to see if the closeness of the results have any relation with the sparseness of the input images.

	Using our vectorization Alg.		Using ZIP Tool	
Input Map (size)	Vector file size	Comp. Rate	Zip file size	Comp. Rate
Figure 5.2-1 (5174bytes)	128 bytes	98%	1005 bytes	81 %
Figure 5.2-2(5174bytes)	352 bytes	93 %	1155 bytes	78 %

Table 5.2-1 The results of compression

5.2 Compression Rates

We also conducted experiments to study the compression rate of our algorithm and compared it to that of ZIP software, a popular general purpose compression tool. The compression rate of our algorithm is measure by the size of the output vector file to the size of the input raster image. We use two raster images of 64 × 64 resolution, the first raster image is a bus route map shown in Figure 5.2-1 and the second raster image is a city map shown in Figure 5.2-2. We compare the compression rates of our algorithm and the ZIP software. The results are shown in Table 5.2.1.

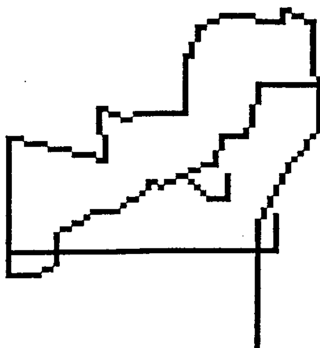


Figure 5.2-1 a bus route map

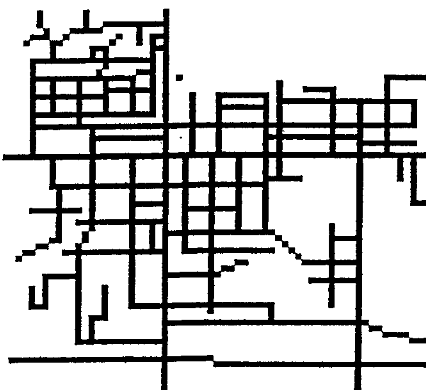


Figure 5.2-2 a city map

The results show that our algorithm gives a better compression rate in both input raster images. As one would have guessed, if the image consists of fewer lines (a sparse image), our algorithm is much better than the Zip tool. When the image becomes dense, the gap between the performance of our algorithm and the Zip tool narrows.

6. CONCLUSION

In this paper we prove that to represent a raster image by a minimum number of line is an NP-complete problem. We also developed an approximation algorithm to solve this problem. Our experiments confirm that the performance of the approximation algorithm is close to the optimal algorithm for raster images that are mainly formed by lines such as bus route maps or city maps.

Our algorithm can serve as a special purpose compression algorithm which generates a portable vector file that can be uncompressed to the original raster image on any devices that use the Bresenham line drawing algorithm. Empirical results show that for some type of raster images, our algorithm has high compression rates and outperforms the general purpose compression utility ZIP tool.

We only use lines as our primitive geometric object in the vector file. It will be interesting to see whether our algorithm can be improved by including more primitive objects such as circles or lines with thickness greater than 1.

REFERENCES

[1] Bresenham, J. E., "Algorithm for Computer Control of a Digital Plotter", *IBM Systems Journal*, Vol. 4, No. 1, pp. 25-30, 1965.

[2] Eastman, "Vector vs. Raster: A Functional Comparison of Drawing Technologies", *IEEE Computer Graphics & Application*, Vol. 10, pp. 68-80, 1990.

[3] Micheal R. Garey, David S. Johnson. "*Computers and intractability*", W.H. Freeman and company, 1979.

[4] Donald Hearn, M. Pauline Baker. "*Computer Graphics*" Prentice Hall, INC. 1986

[5] Parker, J. R. "Extracting Vectors from Raster Images", *Computer & Graphics*, Vol 12, No. 1, pp. 75-79, 1988.