Topological Skeletons

Brooke Dai

CPSC 490

March 29, 2020

Table of Contents

1. Overview 2

    1.1. Properties of an ideal skeleton 2

2. Background 3

    2.1. Blum’s medial axis transform 3

    2.2. Different approaches to skeletonization 7

3. Skeletonization algorithm 8

    3.1. Applying the distance transform 8

    3.2. Identifying a set of ridge point candidates 10

    3.3. Building the skeleton 13

4. Sample problem 17

    4.1. Problem description 17

    4.2. Solution 18

    4.3. Testing method 18

5. References 19

1. Overview

Topological skeletons are what their name suggests: a "skeleton" of a shape that preserves that shape's topology. They serve as a compact description of a shape or object. Beyond this generic definition, there are many other factors to consider depending on the application – the number of dimensions, discrete and continuous cases, whether the input clearly defines which points constitute the object and background, to name a few.

Since skeletons come in many forms and can be seen from many perspectives, I will briefly discuss the major related topics. However, to narrow the scope of this paper, I will be focusing on two-dimensional, discrete points. The input will be in the form of a binary image with 0 representing white background pixels and 1 representing black object pixels.

1.1 Properties of an ideal skeleton

Each skeleton should ideally have the following properties [2]:

Topology: The skeleton should preserve the topology of the shape; the number of holes and components should be reflected in the skeleton.

Thin: The skeleton should be thin.

Centered: The skeleton should be centered within the shape.

Geometric features: The skeleton should preserve the geometric features of the shape; each component of the shape should have a corresponding component on the skeleton.

Reversible: It should be possible to recover the original shape from the skeleton.

It is difficult to find a consensus of a definition that describes skeletons more formally for a variety of reasons. One reason is that there is no one generally agreed-upon definition for some of the properties themselves, such as what exactly "thinness" means [5], or how reasonably reversible a skeleton

should be. This is partly because, depending on the context, different metrics may be considered and different properties may be prioritized, so some definitions may be application-specific. Another reason is that the input is often not perfect and may contain noise on the boundary or within the object, so the skeleton produced also depends on the noise-reduction methods [6].

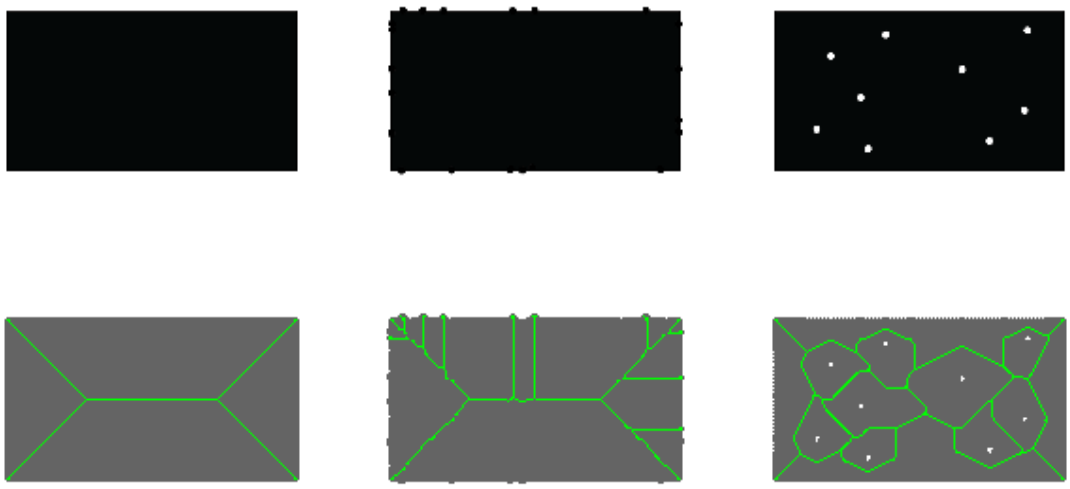

Figure 1.1.1 Skeletons for a rectangle with no noise, a rectangle with noise within the shape, and a rectangle with noise on the boundary.

2. Background

2.1 Blum's medial axis transform

The concept of a *medial axis* (MA) can be seen as the mother of topological skeletons. Harry Blum first coined the term in a paper in 1967 [1], as a new way of describing biological shapes, filling the gaps where previous physics-centric methods of shape analysis had failed. Blum defined the MA as the set of points that make up the corners, or collisions, of wavefronts that propagate from the boundary of a shape in a circular manner with respect to time; the terms will be clarified in the following paragraph.

Consider a continuous plane where each point has three properties: excitation, propagation, and refractory/dead time. Each point can have a value of 1 or 0, corresponding to the point being excited or not (excitation), and each excited point excites the points around it with a delay proportional to the distance between the points (propagation). Once a point is excited, it is not affected by subsequent excitations for an arbitrary amount of time (refractory or dead time). The MA appears when the points on the boundary of the shape are excited and the lines of excited points, or *wavefronts*, collide. The MA with the times of the wavefront collisions included is called the *medial axis function* (MAF).

More informally, the propagation of waves is analogous to a grassfire (and the process is commonly referred to as *grassfire propagation* or the *grassfire transform* [7]*)*. For a shape on a continuous plane, the points inside the shape could be imagined as dry grass. To find the MA of the shape, the boundary of the shape is set on fire and the firefronts close in to the center of the shape. The MA is the set of points where the fire fronts meet and quench each other, and the MAF is the MA with the time of each point's quenching included (Figure 2.1.1).

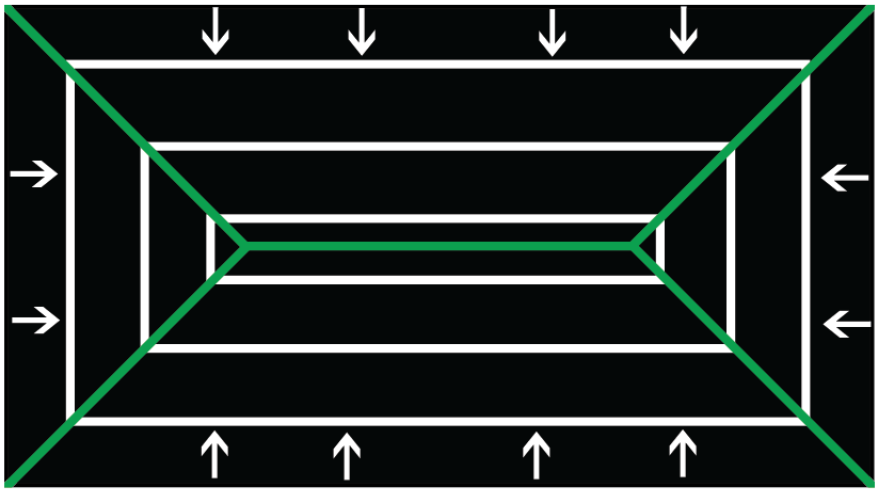

Figure 2.1.1 Obtaining a shape's medial axis through grassfire propagation. The white lines represent the fronts of the fire, and the green lines (the MA) are the points where the fire fronts quench each other.

Given the MAF of a shape, it is possible to reconstruct the shape. This makes comparing shapes with the medial axis transform superior to comparing simple shape descriptors, such as the count of inflection points, that are found purely on the boundary. The MAF carries essential topological properties of the shape that simple shape descriptors do not. Blum demonstrates his point by comparing two different shapes with the same number of inflection points and the same curvature (Figure 2.1.2).

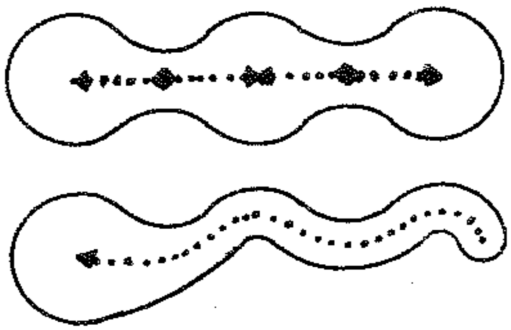

Figure 2.1.2 Two shapes that have the same number of inflection points [1].

There are alternate ways of defining the MAF:

Vertical distance: Instead of time, the MAF is defined by height, and the MA sits on the discontinuity on the surface.

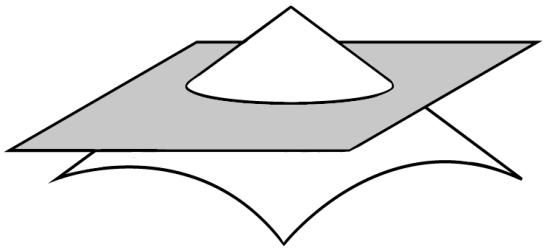

Figure 2.1.3 Each point is given a vertical height, with positive values inside the shape and negative values outside of it.

Boundary distance: The wave is a nearest distance field, ie. each point is assigned the value of its distance to the closest boundary point, and the MAF is made up of the points of discontinuity in the derivative of the field.

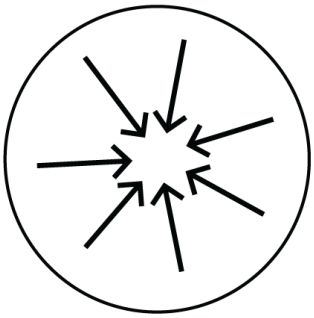

Figure 2.1.4 Arrows pointing in the direction of increasing distance from the boundary point; a discontinuity occurs when the arrows cross the center and switch directions.

Equidistance: Most points in the shape have only one closest boundary point. The MAF is made up of points with two or more boundary points and serves as a line of symmetry in the shape. These points could also be interpreted as the centers of circles that touch the boundary of the shape at two or more places without crossing the boundary.

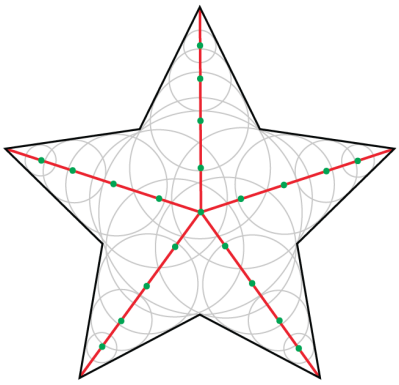

Figure 2.1.5 MA of a star created from the centers of circles that share two or more points with the boundary.

2.2 Different approaches to skeletonization

Blum's work serves as the basis for much of the later research surrounding topological skeletons. However, for many cases, this is the only commonality that can be confidently claimed. The research succeeding Blum's initial findings fragments into many different applications and contexts, often with nuanced differences in definitions and not-so-nuanced differences in approach.

Thinning

Perhaps the most immediately intuitive approach is to iteratively thin, or peel, the shape until no more only the skeleton remains. This approach is subject to restraints on the pixels so that the topology of the shape is preserved. Since every remaining shape pixel must be checked at each iteration, thinning algorithms may be inefficient [4].

Voronoi

Points are sampled on the object's boundary and a Voronoi diagram is created from those points. The skeleton becomes the part of the diagram that intersects with the set of points corresponding to the original shape. As the number of sampled points increases, the resulting skeleton (excluding the *spurious* branches, or extraneous unwanted branches, stemming from the border between two adjacent points) approaches the continuous MA. A downside of this approach is that each point creates an additional branch; as more points are used, more spurious branches are created [2].

Distance transform

The shape undergoes a transformation so that each point on the shape is given the value of its distance to the closest boundary point. The MA is then the ridges of this transformed shape. Each point on the ridge can be seen as the center of maximal inscribed circles in the shape, with the distance value representing the radius of the circle. Different skeletons are produced depending on the distance metric chosen (eg. Manhattan/taxicab, Euclidean, Chebyshev) [8].

3. Skeletonization algorithm

This algorithm uses the distance transform approach and is based on the algorithm described in [3]. The skeleton constructed will be a coarse approximation of the MA. There are three main steps in the algorithm:

1. Apply the distance transform
2. Identify a set of ridge point candidates
3. Build the skeleton from selected ridge point candidates

Each section will provide pseudocode for that section, as well as the names of the relevant functions in the code implementation.

3.1 Applying the distance transform

For each point in the shape, the goal is to find the point's distance to the nearest point on the background. I reference the grassfire transform pseudocode in [7], which uses the Manhattan distance metric, ie. the distance between two points is the number of steps needed to get from one point to the other, where it is only possible to move one unit at a time in either the x- or y-direction. Initially all points have a distance value of 0. Each point considers the four neighbours that it shares a side with, and each point's value on the distance map is equal to one more than the minimum of the distance values of its four neighbours. Let D(x, y) be the value of the point (x, y) on the distance map.

The algorithm takes two passes. The first pass iterates through each point from top to bottom and left to right and only considers the top and left neighbours; that is,

$$D(x, y) = \min(D(x, y - 1) + 1, D(x - 1, y) + 1)$$

for a point (x, y).

The second pass iterates through each point from bottom to top and right to left, and the distance map at the current point (x, y) takes on the minimum of its current value and one more than the distance values of its bottom and right neighbours; that is,

$$D(x, y) = \min(D(x, y), D(x, y + 1) + 1, D(x + 1, y) + 1).$$

The overall effect is as described above: each point's distance value is one more than the minimum distance value of all four of its neighbours (Figure 3.1.1).

0	0	0	0	0	0	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	1	1	1	1	1	0
0	0	0	0	0	0	0

0	0	0	0	0	0	0
0	1	1	1	1	1	0
0	1	2	2	2	1	0
0	1	2	3	2	1	0
0	1	2	2	2	1	0
0	1	1	1	1	1	0
0	0	0	0	0	0	0

Figure 3.1.1 Binary image of a square on the left and its distance map (Manhattan) on the right

Here is the pseudocode, based on the pseudocode from [7]:

```
// First pass
for each row y in image from top to bottom:
      for each column x in image from left to right:
            if (x, y) is in shape:
                  set D(x, y) to min(D(x, y - 1) + 1, D(x - 1, y) + 1)
            else:
                  set D(x, y) to 0
// Second pass
for each row y in image from bottom to top:
      for each column x in image from right to left:
            if (x, y) is in shape:
                  set D(x, y) to min(D(x, y),
                                     D(x, y - 1) + 1,
                                     D(x - 1, y) + 1)
            else:
                        set D(x, y) to 0
```

Each point is visited twice, and the check for the minimum distance value takes constant time. Thus, the complexity of this section is O(image width * image height), or equivalently, O(# pixels).

The code implementation for this step is found in **skeleton.cpp:calculateDistanceMap()**.

3.2 Identifying a set of ridge point candidates

In the discrete case, finding exact ridge points of the distance function can be impossible. For instance, there may be a section of the shape with even width, in which case there will be two adjacent rows (or columns) of points with the same distance value. Chang [3] provides a solution to this problem by introducing the concept of *prominent sign barriers*: a set of patterns of transitions between points that indicate the possible existence of a ridge.

The idea is to find the locations of the discontinuities in the gradient of the distance function; in other words, the local maximum points along lines running through the shape. Two scan lines will be used, one along the x-axis and one along the y-axis, which Chang shows to be sufficient to detect all ridge points, since any ridge lines missed due to being parallel with one scan line will be detected by the other scan line. The distance map is then mapped twice: once as the differences between adjacent points scanned on the x scan line from left to right, which I will call scanX, and once as the differences scanned on the y scan line from top to bottom, which I will call scanY. The equations to find the scan values of a point (x, y) are

$$scanX(x, y) = D(x + 1, y) - D(x, y), \text{ and}$$

$$scanY(x, y) = D(x, y + 1) - D(x, y).$$

The following is pseudocode for finding the distance differences based on the scan lines:

```
for each row y in image from top to bottom:
      for each column x in image from left to right:
            set scanX(x, y) to D(x + 1, y) - D(x, y)
            set scanY(x, y) to D(x, y + 1) - D(x, y)
```

Each value in the scan maps are positive (+), zero (0), or negative (-). Four prominent sign barriers are worthy of note: +-, +0-, +0, and 0-. Each point is labelled in four ways: *strong*, *good*, *weak*, and *none*, corresponding to the strength of the pattern's indication of ridge existence that the point is a part of. Points with the +- or +0- patterns are labelled as *strong*, since the sign change from positive to negative matches immediately to what we are looking for: discontinuities in the gradient. Points with the +0 or 0- pattern are labelled as *weak*, as they do not match the positive-negative pattern that is immediately indicative of a discontinuity; however, when matched with one of the four prominent sign barrier patterns on the other scan line, they are reclassified as *good*. Points with patterns that do not match any of the four prominent sign barrier patterns are labelled as *none*. Chang describes the reasoning behind each labelling in more detail in [3]. The code implementation for this step is in **skeleton.cpp:calculateScanMap()**.

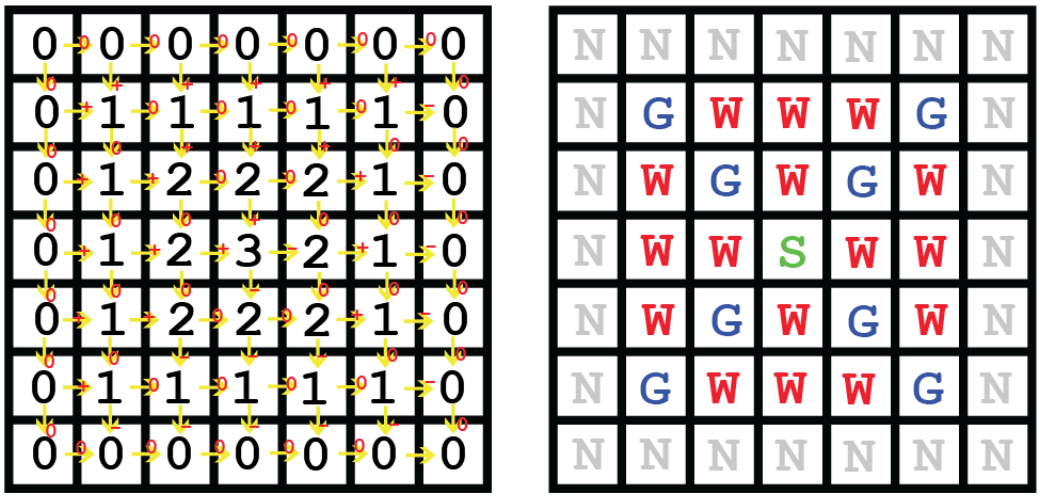

Figure 3.2.1 Each point based on prominent sign barriers in the right image, which are determined from scanning the distance map as shown with the yellow arrows in the left image. (*N - none, W - weak, G - good, S - strong*)

The following is pseudocode for labelling the points:

```
for each row y in image from top to bottom:
      for each column x in image from left to right:
            let prevX be scanX(x - 1, y)
            let currX be scanX(x, y)
            let nextX be scanX(x + 1, y)
            let prevY be scanY(x, y - 1)
            let currY be scanY(x, y)
            let nextY be scanY(x, y + 1)

            // For the x direction, in the case of +-, +0, 0-,
            // prevX is the first sign and currX is the second.
            // in the case of +0-, prevX is the first sign,
            // currX is the second, and nextX is the third.
            // same applies to Y for its respective variables.

            if +- or +0- in the x direction:
                  set label of (x, y) to STRONG
            if +- or +0- in the y direction:
                  set label of (x, y) to STRONG
            if current point is not labelled and
               +0 or 0- in the x direction and
               +0 or 0- in the y direction:
                  set label of (x, y) to GOOD
            if current point is not labelled and
               +0 or 0- in the x direction, or
               +0 or 0- in the y direction:
                  set label of (x, y) to WEAK
            if current point is not labelled:
                  set label of (x, y) to NONE
```

Notice that the scan value of a point is the difference between its own distance value and the previous point's distance value. This choice is largely arbitrary, but should match the checks in the labelling step since we want to choose the point with the maximum distance value.

The actual implementation is more complex since sometimes adjacent points may have the same distance value and can both be labelled as *strong*, but one point is a better choice than the other due to that point already being labelled by the other scan line. Without this check, parallel diagonal lines will appear as in Figure 3.2.1. However, this check cannot be done within the same pass since some essential points may not be labelled at the time of the check. This can be easily remedied by first completing all the *strong* labelling from left to right, then completing the *strong* labelling from top to bottom.

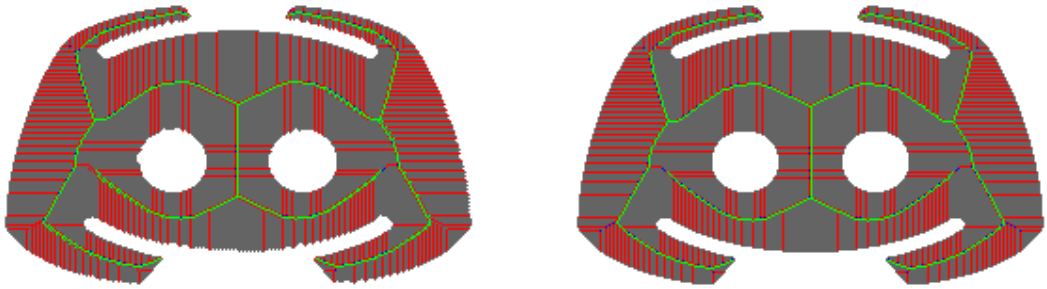

Figure 3.2.1 Ridge point candidates of the Discord logo. The points on the left were determined without the adjacent check mentioned above (note the spotty diagonal lines near the bottom left), the points on the right were determined with the check. The points labelled *strong* are green, *good* blue, and *weak* red. The grey pixels are part of the recreated image based on the ridge point candidates and distance map.

As with the first section, the points are scanned in linear time and the checks and labelling are constant. Thus, the complexity of this section is O(image width * image height), or O(# pixels).

The code implementation for this step is found in **skeleton.cpp:labelCandidates()**.

3.3 Building the skeleton

The final step is to choose the points that will make up the actual skeleton from the set of candidate points. The algorithm takes two passes.

The first pass simply iterates through all the pixels and adds the points labelled *strong* and *good* ridge indicators to the set of skeleton points. The pseudocode is as follows:

```
let S be the set of skeleton points
let visited be array of points already considered for S
for each row y from top to bottom:
      for each column x from left to right:
            if (x, y) is labelled STRONG or GOOD:
                  add (x, y) to S
```

We iterate through every pixel in the image, which takes time linear to the number of pixels. For each pixel, we check its label and add it to the set of skeleton points appropriately, which takes constant time. Therefore, the complexity for this first scan is O(image width * image height), or O(# pixels). The visited array is mainly for the second pass. The code implementation for this step is in **skeleton.cpp:ridgePointsFirstPass()**.

Because we are working with discrete points, just doing one pass is often insufficient as the skeleton can be disconnected. The second "linking" pass aims to connect the various skeleton segments either by repeatedly selecting neighbouring *weak* points, or selecting neighbouring points with the largest distance value, until the skeleton is fully connected. This pass considers all eight neighbours of each point; if only four neighbours are considered, more pixels may be added than desired.

For each point, if the point is in the set of skeleton points and is an endpoint (has only one neighbour) or an isolated point (has no neighbours), tentatively extend a branch in a direction generally opposite the direction of the point's existing neighbour or opposite the direction of the closest boundary point if there are no neighbours. Choose *weak* points where they are available, or points with maximum distance value where there are no *weak* points available. Two cases may result:

1. The tentative branch hits the border of the shape or an invalid pixel, in which case the branch is discarded.

2. The tentative branch hits a skeleton point, in which case the points in the branch are added to the set of skeleton points as a connector between the original endpoint and the newly found skeleton point.

The pseudocode for the second pass is as follows:

```
// defined earlier:
let S be the set of skeleton points
let visited be array of points already considered for S
for each row y from top to bottom:
      for each row x from left to right:
            if (x, y) is in S:
                  let tent be array of points in tentative branch
                  if (x, y) has less than 2 neighbours:
                        let currX be current x-coordinate of tent branch
                        let currY be current y-coordinate of tent branch
                        add (currX, currY) to tent
                        while neither of the two cases are satisfied:
                              check the neighbours of (currX, currY) in
                              the general direction away from existing
                              neighbour or closest boundary point.
                              if there exists a neighbouring point that is
                              in the set of skeleton points:
                                    break
                              if there are no valid neighbours:
                                    set currX and currY to an invalid
                                    point.
                                    break
                              if there exists a WEAK point:
                                    update currX and currY to that point's
                                    coordinates
                              else:
                                    Update currX and currY to the
                                    coordinates of the neighbour with the
                                    largest distance value
                              if (currX, currY) is in visited:
```

```
                    break
              else:
                    add (currX, currY) to visited
        if (currX, currY) is invalid or not on the shape:
              continue
        for each (arrX, arrY) in tent
              add (arrX, arrY) to S
```

We iterate through each point in the image, which takes time linear to the number of points in the image. For each point, if the point is in the skeleton set, we count the number of neighbours it has, which takes constant time. If the point has less than two neighbours, we extend a tentative branch by selecting *weak* points or points with a maximum distance value that have not yet been considered for adding to the skeleton set. Since the visited values persist with each tentative branch, we can be sure that the number of points visited by all tentative branches is linear in respect to the number of pixels. If the tentative branch successfully finds another point in the skeleton set, each point in the tentative branch is added to the skeleton set, which again takes linear time. Thus, the complexity for the second scan is O(image width * image height), or O(# pixels). The code implementation for this step is found in **skeleton.cpp:ridgePointsSecondPass()**.

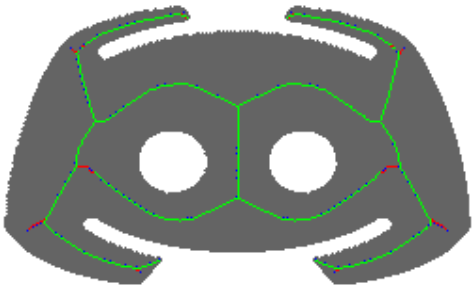

Figure 3.3.1 Final skeleton of the discord logo.

Each part of the overall skeletonization algorithm takes linear time, and when they are combined the complexity remains linear. Therefore, the overall time complexity of the skeletonization algorithm is O(image width * image height), or O(# pixels).

4. Sample problem

4.1 Problem description

Suppose that you are a scout in a second-rate crime organization, tasked to collect information about the underground tunnels of a prestigious bank vault. Your task is to send an accurate map of the tunnels with the correct dimensions (in 2D, looking from the top down).

However, you are only given a cell phone as a device to collect information about the tunnel dimensions. Doubly insulting is the fact that, since the organization isn't top tier, the cell phone you are given has limited capabilities - namely, that it can only send numbers and not images or actual scans to your organization's network; anything other than numbers and it catches on fire and explodes.

Your boss, being sympathetic to the situation, devises a plan so that you can send all the necessary information while not having the phone explode. He gives you a laser pointer, which can give you the Manhattan distance between yourself and any wall you point it at, and tells you to send a set of tuples (x, y, d) that can be used to reconstruct the tunnels. A Manhattan circle will be drawn at each x, y with radius d. The reconstructed map will then be the union of all such circles.

Your boss, being the kind person that they are, also tells you that you are allowed an error of a one pixel boundary around the boundary of the tunnels, but if the crime lord finds that the error is greater, you will be unceremoniously thrown into the sewers.

Input

The first line contains two integers W and L, $1 \leq W \leq 10^7$, $1 \leq L \leq 10^7 / W$, representing the width and length of the rectangle in which the tunnels are contained (in meters).

L lines follow. Each line will contain W values of integers in [0, 1], 1 representing tunnel space and 0 representing the ground around the tunnel. The coordinate (0, 0) is the top left corner.

Output

Print an integer N on the first line corresponding to the number of tuples you will send.

On each of the next N lines, print x, y, and d, where (x, y) is the coordinate of the center of the Manhattan circle of radius d that will be drawn in the reconstructed map. It is possible that the tunnels do not exist.

4.2 Solution

Using the skeletonization algorithm described above, we can build a skeleton of the tunnels, keeping track of the distance value of each point. Since the distance value of a point can be seen as the radius of a circle with that point as the center, we can output the set of skeleton points and their distance value.

4.3 Testing method

The set of points from the output are converted to another image, which is then compared to the first image. The output is valid if the following conditions hold:

1. No point on the reconstructed image has a value of 1 where there should be a value of 0;
2. The tunnels are accurate to 1 pixel from the border (ie. all pixels with distance value 1 in the original image can have value 0 in the reconstructed image, and all pixels with distance value 0 can have value 1 in the reconstructed image).

5. References

1. Blum, H. “A transformation for extracting new descriptors of shape.” *Models for the Perception of Speech and Visual Form,* Weiant Wathen-Dunn, MIT Press, 1967, pp. 362-380
2. Saha, Punam K., et al. Skeletonization: Theory, Methods and Applications. Academic Press., 2017, Chapter 1.
3. Chang, S. “Extracting Skeletons from Distance Maps.” IJCSNS International Journal of Computer Science and Network Security, VOL.7 No.7, July 2007, pp. 213-219.
4. Vincent, L. “Efficient computation of various types of skeletons.” Proc SPIE Vol. 1445, “Medical Imaging V”, San Jose CA, February 1991, pp. 297 - 311.
5. Zhang, T. Y., and C. Y. Suen. “A Fast Parallel Algorithm for Thinning Digital Patterns.” Communications of the ACM, vol. 27, no. 3, Jan. 1984, pp. 236–239, doi:10.1145/357994.358023.
6. Fisher, R., et al. “Skeletonization/Medial Axis Transform.” Morphology - Skeletonization/Medial Axis Transform, homepages.inf.ed.ac.uk/rbf/HIPR2/skeleton.htm.
7. Wikipedia contributors. "Grassfire transform." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 27 Jan. 2020. Web. 30 Mar. 2020.
8. Wikipedia contributors. "Distance transform." Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 3 Dec. 2019. Web. 30 Mar. 2020.