I like trees because they seem more resigned to the way they have to live than other things do.

—Willa Cather

5.1 OVERVIEW
We have so far discussed the parametric active contour and geometric active contours that are realized via energy minimization with continuous optimization techniques. The focus of this section is to illustrate a few renowned discrete optimization techniques for image segmentation based on graphs. Unlike the continuous optimization framework, which typically aims at finding a local minimum in energy, these graph algorithms find globally optimal solutions. Also, the running time for such algorithms is an important consideration.

Before we delve into these fascinating techniques, first let us familiarize ourselves with the proper terminology in this context. If the variables in an optimization method are restricted to take only discrete values, the optimization is called a discrete optimization. When the underlying data structure for a discrete optimization is a graph, we typically refer to it as a graph algorithm. Examples include shortest path, minimum spanning trees, minimum cut computation, and so on. In this section, we will use graph algorithms for two types of computation:

1. Contour or path computation based on shortest path (SP) techniques
2. Region-based segmentation via graph cuts.

5.2 SHORTEST PATH SNAKES
The motivation behind the SP snakes can be explained in terms of user interactive means of object boundary/contour delineation. Suppose we want to quickly delineate an outline in a magnetic resonance spine image. Instead of drawing a closed contour by the mouse cursor, we certainly would like to use a few mouse clicks and expect that the majority of the boundary delineation task will be
performed automatically after this user interaction. If a few places of the object boundary contour need correction, we would like to do that at a latter stage, where, again, we do not want to spend much time or effort. This kind of image analysis tasks can be performed quite well with the SP snakes. The same task can also be made automatic with SP snakes by using a trick to be detailed shortly.

Before making some headway into the segmentation algorithm, some requisite preliminaries about graphs and SP are laid out here. A graph $G$ is comprised of two sets—a set of vertices $V$ and a set of edges $E$. It is commonly denoted as a pair $G = (V,E)$. Figure 5.1a illustrates a graph. For this graph, the vertices are as follows: $V = \{1, 2, 3, 4, 5\}$. The edges in a graph are the connections among the vertices. For the graph in Figure 5.1a, the set of edges is: $E = \{(1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (4, 5)\}$. Figure 5.1b and 5.1c show two graphs on a rectangular image domain. The first one is a four-connected graph and the second one is an eight-connected graph.

A path is a collection of connected edges between a source vertex and a destination vertex. For example, in Figure 5.2, between the source vertex A and destination vertex B, an example path is $\{(A, C), (C, D), (D, B)\}$. Another path is $\{(A, E), (E, D), (D, B)\}$. Further, we can attach weights to the edges in a graph. For the example in Figure 5.2, the numbers labeling edges signify the edge weights. Thus, on an edge-weighted graph, a path will have a weight, too. In the previous example, the path $\{(A, C), (C, D), (D, B)\}$ has a weight $2 + 4 + 1 = 7$, whereas the path $\{(A, E), (E, D), (D, B)\}$ has a weight 5. If these edge weights define some kind of distance between two vertices, then we would interpret that the latter path is shorter than the former. In this sense, we refer to the weight of a path as the length of that path. In a general setting, the length/weight of an edge can be set any real number—positive, negative, or zero. Note that for this example graph in Figure 5.2, we can interchangeably use the terms source and destination, i.e., there is no harm in interchanging the source and the destination vertices as the paths remain the same between them.
In the SP problem, given a source vertex $s$ and a destination vertex $t$ in a graph, we want to find out the shortest path $P$ between them. There are some variations in the shortest path problem. The one stated here is called the point-to-point shortest path. For one of the most recent advances on the point-to-point SP problem, see Reference [48]. The two classical versions of the problem are one source to all destinations, and all source to all destinations. The classical algorithm for one source to all destinations is Dijkstra’s algorithm [49], which restricts the weight/length of the edges to be positive.

Let us now turn our attention from the graph and SP preliminaries to the snake computation. We can think of two user chosen points as the source and the destination vertices on the image domain. Now, the problem is to find an SP between the source and the destination points, so that the SP defines the object boundary. It should now be obvious that if the SP needs to be the object boundary, we require an effective edge weight function. For the images shown in Figure 5.3, image gradient magnitude seems to be a dominant cue for delineating the object boundary. Thus, we define the cost here as:

$$D(i, j) = \exp \left( - \frac{|\nabla I(i)| + |\nabla I(j)|}{2w} \right),$$

(5.1)

where $i$ and $j$ are any two adjacent pixels on the image domain. $w$ is a positive user defined parameter. For an eight-connected graph, we need to define these costs for all eight pairs for every interior pixel. For the image border pixels, we define these costs for their available neighbors as well. Figure 5.3 shows the result of interactive segmentation. $D(i, j)$ is essentially a distance between two neighboring pixels $i$ and $j$. Note that if $i$ and $j$ are object edge pixels, the image gradient magnitude will be large at their locations, and accordingly, the distance between them will be smaller. On the other hand, if both of $i$ and $j$ are not object edge pixels, or if just one of them is not an object edge pixel, then the distance between them will be not be as small. To illustrate, we used Dijkstra’s SP algorithm for computing SP in Figure 5.3.
Of course, it is desirable to use the SP approach without user interaction in some scenarios. Here, the source and destination pixels are supplied automatically in some way. Some additional tricks are usually necessary. One approach involves the creation of barriers. To elucidate the idea, let us take a quick look at the graph in Figure 5.1a. If we assume all the edges in this graph have the same weight, the SP from vertex 3 to vertex 4 is the direct link (3,4) between them. However, if this edge (3,4) did not exist for this graph, then the SP from 3 to 4 would be {(3,2), (2,5), (5,4)}. Removal of an edge between two vertices thus creates a barrier between them. This same approach can be taken with the graph formed by the image domain grid (Figure 5.1b or 5.1c). For example, as illustrated in Figure 5.4, from the image grid, we have removed the pixels marked by the vertical and horizontal lines. Vertex removal essentially creates barriers in the image grid because all the edges attached to these vertices are also removed. Let us now consider two pairs of pixels (A,B) and (C,D) as shown in Figure 5.4. With the aforementioned barriers, the SP snake computed between A and B is shown in cyan, while that between C and D is shown in yellow in Figure 5.4. These barriers and the points A, B, C, and D are all chosen automatically with some preliminary processing of the MR slice. Note that barrier-induced SP snakes can also be computed in the user interactive setting.

Although the results in Figures 5.3 and 5.4 are impressive, we must discuss the limitations of SP snakes. One severe limitation of the SP paradigm is inherent in the cost function Equation 5.1. The cost/distance is defined only for an edge in the graph, i.e., only for a pair of adjacent pixels in the image domain. If we desired a smoother object contour, we would be prohibited to drift from the existing edges. So, if the image is noisy, without a proper cost function, SP snake can produce
contours with irregularities. However, even in such cases, SP snakes can provide a nice initialization for other snake optimizations such as dynamic programming or gradient descent that can straighten the contours.

5.3 BINARY LABELING WITH GRAPH CUT
Labeling the pixels in the image domain as foreground or background is referred to as binary labeling. We have already seen examples of this in Chapter 4 from a continuous energy minimization perspective. In this section, we will illustrate two well-known discrete optimization approaches, both based on graph algorithms ([50], [51]).

Once again, we will view the image domain as a connected graph, i.e., a collection of connected vertices as shown in Figure 5.1b or 5.1c. The basic principle behind graph-based binary pixel
Labeling is to divide the image domain into two different sets—label some pixels as 0 (background) and the rest as 1 (foreground). The technical name for this procedure is called *graph cut*. A cut for a graph \( G = (V, E) \) with vertex set \( V \) and edge set \( E \) is formed by removing a subset \( R \) of edges from \( E \). This removal of the edges creates two graphs \( (V_1, E_1) \) and \( (V_2, E_2) \), so that there is no edge between the vertex sets \( V_1 \) and \( V_2 \). In Figure 5.5, we illustrate a cut of the four-connected image domain grid of Figure 5.1b. Note that Figure 5.5 contains two graphs—one with hollow pixels (say, background) and the other with solid pixels (say, foreground). These two graphs are the results of a cut, i.e., removal of a set \( R \) of edges from Figure 5.1b. The hollow and the solid pixels are merely symbolic and used to denote the binary labeling that resulted from this graph cut.

Central to any graph cut algorithm is the concept of the cost of a cut defined as:

\[
\text{cost}(V_1, V_2) = \sum_{(i,j) \in R} W(i, j),
\]  

(5.2)

i.e., the cost of a cut is the sum of the cost of all the removed edges. There are various ways to obtain a cut of a graph based on the cost of the cut. Here, we will discuss two well-embraced methods—minimum cut (mincut) and normalized cut (Ncut).

### 5.4 Binary Labeling with Minimum Cut

Binary pixel labeling is often performed by minimizing a cost function of the following form:

\[
E(p_1, \ldots, p_N) = \sum_{i=1}^{N} f_i(p_i) + \sum_{(i,j) \in \Omega} g(i,j)(p_i, p_j),
\]  

(5.3)

where \( p_1, \ldots, p_N \) are binary pixel labels to be determined, i.e., \( p_i \in \{0, 1\} \), \( N \) is the total number of pixels in the image domain, \( f_i \) is a function that can possibly depend on the pixel location \( i \) and model the foreground and the background, \( g(i,j) \) is a function that typically imposes smoothness in the
solution. For example, \( g(i, j) \) will penalize if two neighboring pixels \( i \) and \( j \) have different labels: \( p_i \neq p_j \). \( \Omega \) is the set of neighboring pixel pairs \( (i, j) \). In the set \( \Omega \), the order of vertices is not distinguished: the pixel pairs \( (i, j) \) and \( (j, i) \) are considered the same. Functions of this form (Equation 5.3) are sometimes referred to as a Markov random field (MRF) model [50].

The mincut method can be applied to obtain a globally minimum solution for Equation 5.3 provided the function \( g(i, j) \) obeys a regularity condition (not to be confused with regularization!) [50]:

\[
g(i, j) (0,0) + g(i, j) (1,1) \leq g(i, j) (1,0) + g(i, j) (0,1) \quad \text{for all neighbors } (i, j) \in \Omega.
\]

The trick for minimizing Equation 5.3 via graph cut is to first represent the cost function Equation 5.3 as the cost of a cut of a suitable graph. Then, obtain the minimum cut by efficient graph algorithms (see Reference [50] and references therein). Here, we do not discuss implementations of any minimum cut algorithm; rather, we focus on graph-cut representation of the cost function Equation 5.3.

In what follows is a graph construction procedure adopted from Reference [52]. We build a graph \( G \) with \( N + 2 \) vertices of which \( N \) vertices denote the \( N \) pixels, and the rest of the two vertices are special ones. One of them is called a source vertex \( s \), and the other one is called a destination/sink vertex \( t \). Among the \( N \) pixel-vertices there are edges between each neighboring pairs (four neighbors or eight neighbors, as the case may be). There are edges from source vertex \( s \) to the pixel-vertices. Also, there are edges from the pixel-vertices to the sink vertex \( t \). We refer to such a graph as a source-sink graph. An example source-sink graph is shown in Figure 5.6a, where a small four-pixel (four-connected) image along with a source vertex \( s \) and a sink vertex \( t \) are seen. We discuss shortly how to set these edge weights in a source-sink graph. A cut on this graph is defined to produce two graphs, so that one graph contains the source vertex \( s \) and the pixel-vertices with label \( p_i = 0 \) and the other graph with a vertex set comprising of the sink vertex \( t \) and the pixel-vertices with label \( p_i = 1 \). In set notations, these two vertex sets are, respectively, \( \{ s \} \cup \{ i : p_i = 0 \} \) and \( \{ t \} \cup \{ i : p_i = 1 \} \). An example cut of a source-sink graph is shown in Figure 5.6b. The aim here is to set

**FIGURE 5.6:** (a) A source-sink graph. (b) A cut of the graph in (a).
the edge weights in the source-sink graph \( G \) in such a way that the energy function Equation 5.3 is represented by the cost of a cut of \( G \).

To represent the first component of the energy function Equation 5.3 within the cost of a cut of \( G \), it is written as:

\[
\sum_{i=1}^{N} f_i(p_i) = \sum_{i} [p_i f_i(1) + (1 - p_i) f_i(0)] = \sum_{i} p_i[f_i(1) - f_i(0)] + \sum_{i} f_i(0)
\]

\[
= \sum_{i} p_i \max(0, f_i(1) - f_i(0)) + \sum_{i}(1 - p_i) \max(0, f_i(0) - f_i(1))
\]

\[
+ \sum_{i} [f_i(0) - \max(0, f_i(0) - f_i(1))].
\] (5.4)

In deriving the last equality, we have used an identity that for any real number \( z \): \( z = \max(0, z) - \max(0, -z) \). To represent Equation 5.4 in the source-sink graph \( G \), we add an edge from the source vertex \( s \) to the pixel vertex \( i \) with weight \( f_i(1) - f_i(0) \) if \( f_i(1) - f_i(0) > 0 \); else, if \( f_i(1) - f_i(0) < 0 \), we add an edge from \( i \) to \( t \) with weight \( f_i(0) - f_i(1) \). So, if \( f_i(1) > f_i(0) \), then there will be an edge \((s, i)\), in \( G \) and removal of this edge will cost \( f_i(1) - f_i(0) \). On the other hand, if \( f_i(1) < f_i(0) \), then there will be an edge \((i, t)\) in \( G \). The cost of removing this edge is \( f_i(0) - f_i(1) \). Note that the last summation term in Equation 5.4 does not involve pixel labels and is essentially a constant. Thus, this term need not be represented in the cut cost.

To realize the second term in Equation 5.3, viz., \( \sum_{(i,j) \in \Omega} g(i,j)(p_i, p_j) \) within the cut cost, we rewrite it as follows:

\[
\sum_{(i,j) \in \Omega} g(i,j)(p_i, p_j) = \sum_{(i,j) \in \Omega} [p_i p_j g(i,j)(1,1) + p_i(1 - p_j) g(i,j)(1,0)
\]

\[
+ (1 - p_i) p_j g(i,j)(0,1) + (1 - p_i)(1 - p_j) g(i,j)(0,0)]
\]

\[
= \frac{1}{2} \sum_{(i,j) \in \Omega} (p_i - p_j)^2 [g(i,j)(1,0) + g(i,j)(0,1) - g(i,j)(1,1) - g(i,j)(0,0)]
\]

\[
+ \frac{1}{2} \sum_{(i,j) \in \Omega} p_i [g(i,j)(1,1) + g(i,j)(1,0) - g(i,j)(0,1) - g(i,j)(0,0)]
\]

\[
+ \frac{1}{2} \sum_{(i,j) \in \Omega} p_j [g(i,j)(1,1) + g(i,j)(0,1) - g(i,j)(1,0) - g(i,j)(0,0)]
\]

\[
+ \sum_{(i,j) \in \Omega} g(i,j)(0,0). \] (5.5)

From Equation 5.5, it is clear that we add an edge between every neighboring pixel pair \((i,j)\) with weight \( 0.5[g(i,j)(1,0) + g(i,j)(0,1) - g(i,j)(1,1) - g(i,j)(0,0)] \). For the second summa-
segmentation term, if \[ g(i, j) (1,1) + g(i, j) (1,0) - g(i, j) (0,1) - g(i, j) (0,0) > 0 \], we add an edge from \( s \) to \( i \) with weight \( 0.5[g(i, j) (1,1) + g(i, j) (1,0) - g(i, j) (0,1) - g(i, j) (0,0)] \); else, if \[ g(i, j) (1,1) + g(i, j) (1,0) - g(i, j) (0,1) - g(i, j) (0,0) < 0 \], we add an edge from \( i \) to \( t \) with weight \( 0.5[g(i, j) (0,0) + g(i, j) (0,1) - g(i, j) (1,0) - g(i, j) (1,1)] \). Similarly, for the third summation term, we add an edge from \( s \) to \( j \) if \[ g(i, j) (0,0) + g(i, j) (0,1) - g(i, j) (1,0) - g(i, j) (1,1) \]; else, if \[ g(i, j) (0,0) + g(i, j) (0,1) - g(i, j) (1,0) - g(i, j) (1,1) \] < 0 we add an edge from \( j \) to \( t \) with weight \( 0.5[g(i, j) (0,0) + g(i, j) (0,1) - g(i, j) (1,0) - g(i, j) (1,1)] \). If one of these edges exists from previous edge constructions, the existing edge weight is incremented by the current weight.

Note that once again nothing needs to be done for the fourth summation term in Equation 5.5 as it is essentially a constant. For a slightly different graph construction, see also Reference [50]. Thus, a cut of \( G \) now represents the energy/cost function Equation 5.3 up to an additive constant. Having constructed the source–sink graph \( G \), one obtains the minimum cut, which minimizes the energy function Equation 5.3 and provides the corresponding binary labeling of the image.

Now, we illustrate the mincut on an image shown in Figure 5.7a. Note the cells in Figure 5.7a have essentially two dominant intensity levels, where as the background has practically a single dominant intensity level. Now, we can define the following Gaussian probability models for a pixel \( i \):

\[
P(p_i = 0) = \frac{1}{\sqrt{2\pi}\sigma_b} \exp \left( -\frac{(I(i) - \mu_b)^2}{2\sigma_b^2} \right),
\]

\[
P(p_i = 1) = \frac{0.5}{\sqrt{2\pi}\sigma_{f,1}} \exp \left( -\frac{(I(i) - \mu_{f,1})^2}{2\sigma_{f,1}^2} \right) + \frac{0.5}{\sqrt{2\pi}\sigma_{f,2}} \exp \left( -\frac{(I(i) - \mu_{f,2})^2}{2\sigma_{f,2}^2} \right),
\]

where \( \mu_b \) and \( \sigma_b \) stand respectively for the mean and the standard deviation of the background intensity. The foreground is composed principally of two intensity levels, denoted by \( \mu_{f,1} \) and \( \mu_{f,2} \), along with respective standard deviations \( \sigma_{f,1} \) and \( \sigma_{f,2} \). The parameters of these models needs to be

![Figure 5.7](image-url)
estimated (say, by maximum likelihood estimation) from training images. Now, we are in a position to define the function \( f \) in Equation 5.3:

\[
    f_i(1) = -\ln(P(p_I = 1)) \quad \text{and} \quad f_i(0) = -\ln(P(p_I = 0)).
\]

We can also define the function \( g \) in Equation 5.3 as follows:

\[
    g(i, j) (0,0) = g(i, j) = 0, \quad \text{and} \quad g(i, j) (1,0) = g(i, j) (0,1) = \lambda
\]

where \( \lambda \) is a positive number, essentially a tuning parameter that controls the balance between the two summation terms in Equation 5.3. With such a model, we segmented the image in Figure 5.7a by minimum cut to produce the result shown in Figure 5.7b).

### 5.5 Pixel Labeling with Normalized Cut

In the previous section, we have seen the action of mincut toward binary labeling. In this section, we will see the act of another graph cut criterion—normalized cut, also known as Ncut. The Ncut criterion has been successfully used in the past for grouping/clustering in computer vision. The basic question grouping/clustering tries to answer is this: given a number of objects \( N \) and a number of groups/clusters \( K \), how to put \( N \) objects into \( K \) groups. The connection to image segmentation should be immediately clear: how to assign labels to image pixels. Obviously, in this setting, one needs to define the affinity or similarity between two objects, in this case, two pixels. The following is an example of affinity between two pixels \( i \) and \( j \) in an image \( I \):

\[
    W(i, j) = \exp \left( -\frac{|I(i) - I(j)|^2}{2\sigma_I^2} \right) \exp \left( -\frac{|X(i) - X(j)|^2}{2\sigma_X^2} \right), \quad (5.6)
\]

where \( X(i) \) denotes the coordinates of the pixel \( i \) in the image domain. So, \( |X(i) - X(j)| \) denotes the distance between two pixels \( i \) and \( j \). Note that the affinity between two pixels is more when they are closer to each other both in terms of intensity and locations in the image grid. The \( \sigma \)'s appearing in Equation 5.6 are scaling parameters that the user must supply. Note that in defining the affinity matrix \( W \), \( i \) and \( j \) pixels need not be four or eight neighbors. The matrix can be defined for every pixel pair. The affinity matrix \( W \) in essence defines a graph for the image domain wherever a pixel is connected with every other pixel. The edge weights in the graph are specified by the elements of \( W \). Such a graph is called a fully connected graph.

Ncut is a grouping/clustering technique where the clustering can typically be performed either in a single shot or in a hierarchical fashion. In the single shot clustering paradigm, one assigns \( N \) pixels to \( K \) classes or clusters all pixels in one step. When \( K = 2 \), we have binary clustering. In the hierarchical clustering, we essentially form a binary tree of objects by performing repeated binary clustering. Then, the tree pruning can be followed to merge groups. Here, we will only discuss the binary clustering.
One way to perform a binary clustering/labeling is to consider the minimum cut of the fully connected graph induced by the affinity matrix $W$. However, after playing with an image with min-cut on $W$, one will soon realize that even with a nicely custom-designed cost matrix $W$, the minimum cut we obtain is quite useless. This is because the minimum cut we typically obtain in such a case is highly imbalanced: we get a very large set of pixels labeled 0 (or 1) and a very small set of pixels labeled 1 (or 0). It might be as bad as one pixel in one set versus the remainder in the other set. This result is not surprising because mincut merely minimizes the total cost of the removed edges. From the segmentation perspective, such imbalanced labeling is of no practical use. A thought that might have left the reader perplexed at this point is why we did not get an imbalanced labeling with minimum cut on the cost function Equation 5.3. The answer to this apparent contradiction is that Equation 5.3 has both an affinity-based term (the second summation) as well as a region-term (the first summation), so the mincut produced acceptable results.

Our naïve effort with minimum cut on the affinity matrix $W$ leads to the following questions: what other meaningful criteria exist to obtain a somewhat balanced cut? and computationally speaking, how efficient are these cuts? The Ncut criterion achieves some balance between the two cuts. Resorting to our previous notations, if a cut of a graph $G = (V, E)$ creates two graphs $(V_1, E_1)$ and $(V_2, E_2)$ by removing a set of edges $R$ from $E$, then Ncut is defined as:

$$\text{Ncut}(V_1, V_2) = \frac{\text{cost}(V_1, V_2)}{\text{assoc}(V_1, V)} + \frac{\text{cost}(V_1, V_2)}{\text{assoc}(V_2, V)},$$  \hspace{1cm} (5.7)

where ‘assoc’ is defined as:

$$\text{assoc}(V_1, V) = \sum_{(i,j) \in E_1 \cup R} W(i, j),$$

$$\text{assoc}(V_2, V) = \sum_{(i,j) \in E_2 \cup R} W(i, j).$$

Above, ‘assoc’ is the association cost of a vertex set $V_1$ (or $V_2$) with the original vertex set $V$. Note that the function “cost” is already defined via Equation 5.2. This Ncut cost Equation 5.7 normalizes a cut cost with respect to the association cost. In fact, it is easy to see that each ratio in Equation 5.7 will never exceed 1; thus, the Ncut value lies between 0 and 2. Normalized cut method minimizes the Ncut cost Equation 5.7 to produce a cut of the fully connected image domain. This cut is essentially a binary labeling of the image.

Let us now examine what happens to the Ncut value when a cut is highly imbalanced. For an imbalanced cut with size of $V_1$ much larger than that of $V_2$, $\text{assoc}(V_1, V)$ will be comparable to cost $(V_1, V_2)$, while $\text{assoc}(V_2, V)$ will be much larger than cost$(V_1, V_2)$, bringing the Ncut value close to 1, which is still an appreciable value, and thus, an unbalanced cut will most likely be not the minimum of Equation 5.7.
With some algebraic manipulations to Equation 5.7, Shi and Malik showed that minimizing Equation 5.7 is equivalent to finding out the eigenvector corresponding to the second smallest eigenvalue of the generalized eigensystem [51]:

\[(D - W)p = \lambda p,\]

where \(D\) is a diagonal matrix with diagonal element:

\[D(i, i) = \sum_{j=1}^{N} W(i, j), \forall i = 1, \ldots, N.\]

The second smallest eigenvector \(p\) (an \(N\)-by-1 column vector) of the system Equation 5.8 provides a minimum of Equation 5.7.

In Figure 5.8a, we show a blood cell image that we want to segment (binary label) with Ncut. The affinity matrix we consider for this problem is:

**FIGURE 5.8:** (a) A blood cell image. (b) Eigenvector corresponding to second smallest eigenvalue. (c) Binary labeling via Otsu’s method. (d) Eigenvector corresponding to third smallest eigenvalue. (e) Ternary labeling via k-means clustering.
\[ W(i, j) = \begin{cases} \exp \left( -\frac{|I(i) - I(j)|^2}{2\sigma_I^2} \right) \exp \left( -\frac{|X(i) - X(j)|^2}{2\sigma_X^2} \right), & \text{for } |X(i) - X(j)| \leq 25. \\ 0, & \text{otherwise.} \end{cases} \] (5.9)

Note that \( W \) in Equation 5.9 is a sparse matrix, whereas the \( W \) in Equation 5.6 is a dense one. Creating a sparse matrix helps speed up the computation of the eigenvector in Equation 5.8. Figure 5.8b shows the second smallest eigenvector \( p \) of the system Equation 5.8 after we transform the column vector \( p \) to a two-dimensional matrix by the column-major order. Notice that minimization (Equation 5.7) should lead us to a binary labeling; however, the matrix in Figure 5.8b is not a binary one. So, what happened? It turns out that minimizing Equation 5.7 is approximately equivalent to finding out the second smallest eigenvector of Equation 5.8 because there is no guarantee that this eigenvector of Equation 5.8 will be a binary vector. There is a very good computational reason behind this approximation. The reason being that finding a binary solution of Equation 5.7 is an NP-complete problem [51]. For those who are unfamiliar about the notion of NP-completeness [53], it suffices for now to say that finding efficient polynomial time algorithms for these problems has not yet been possible, even with many decades of computer science research. Practically speaking, we can obtain a binary labeling from Figure 5.8b by applying Otsu's thresholding method [54]. This yields the binarized image of Figure 5.8c.

Let us close this discussion by showing that Ncut can, in fact, produce multivalued labeling. For example, we can perform a ternary labeling of the blood cell image in Figure 5.8a by the Ncut method. In this case, along with the second smallest eigenvector, we also need the third smallest one (shown in Figure 5.8d). Next, we need a clustering method such as \( k \)-means [55] to produce the ternary labeling shown in Figure 5.8e. The input to the \( k \)-means algorithm is both the eigenvectors shown in Figure 5.8b and 5.8d.