Segmentation with Graph Algorithms

We have so far discussed the parametric active contour and geometric active contours from energy minimization with continuous optimization techniques. The focus of this section is to illustrate a different aspect of snake and related computing. We will discuss a few renowned discrete optimization techniques on graphs applied to image segmentation. Unlike the continuous optimization framework, which typically aims at finding a local minimum, these graph algorithms find globally optimum solutions. Also the running time for such algorithms is an important consideration.

Before we delve into these fascinating techniques, first let’s 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, and
(2) Region based segmentation via graph cuts.

Shortest Path Snakes

Perhaps the motivation for the shortest path snakes can be best explained in terms of user interactive means of object boundary/contour delineation. Suppose we want to quickly delineate the gray matter of a CT 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 shortest path (SP) snakes. The same task can also be made automatic with SP snakes by using a trick shown shortly.

![Graphs and image grids](image)

Fig 1: (a) A graph. (b) 4-connected image grid. (c) 8-connected image grid.

Before making our headway, some absolute preliminary about graphs and shortest path (SP) is laid out here. A graph $G$ comprises of two sets– a set of vertices $V$ and a set of edges $E$. It is commonly denoted as a pair $G = (V, E)$. Fig. 1(a) illustrates a graph. For this graph the vertices and the edges are as follows: $V = \{1, 2, 3, 4, 5\}$ and $E = \{(1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (4, 5)\}$. Thus edges are the connections among the vertices in a graph. Fig. 1(b) and 1(c) show two graphs on a rectangular image domain. The first one is a 4-connected graph and a second one is an 8-connected graph, except for the border pixels where connectivity is less than 4 and 8 respectively.
A path is a collection of connected edges between a source vertex and a destination vertex. For example, in Fig. 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 example in Fig. 2 the numbers 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 of 2+4+1=7, whereas the path \{(A, E), (E, D), (D, B)\} has a weight of 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 Fig. 2, we can interchangeably use the terms source and destination, i.e., there is no harm in confusing between the source and the destination vertices because 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 would like to find the shortest path \(P\) between them. There are different versions of the shortest path problem. The one stated here is called P2P or point to point shortest
path. For one of the most recent advances on P2P problem see [GH05]. 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 [CLRS] that expects 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 mouse clicks as the source and the destination
vertices on the image domain. Now the problem is to find an SP between the two mouse
clicks, 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 Fig. 3, image gradient magnitude seems to be a dominant cue for
delineating the object boundary. Thus we define the cost here as:

\[ D(i, j) = \exp\left(-\frac{|\nabla I(i)| + |\nabla I(j)|}{2w}\right), \]  
\[ (1) \]
where \( i \) and \( j \) are any two adjacent pixels on the image domain, \( i.e., \) there is an edge between the pixel \( i \) and the pixel \( j \). 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. Fig. 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 even any one of them is not an object edge pixel, then distance between them will be not as small. We used Dijkstra’s SP algorithm for computing SP in Fig 3.

Inquisitive readers should be wondering if SP snake can be utilized without user interactions. Obviously, if the source and destination pixels are supplied automatically in some way, then SP snake can be used as an automatic segmentation tool. However, often in the automatic setting we need additional tricks. We refer to this trick as creating *barriers*. To elucidate the idea, let us take a quick look at the graph in Fig. 1(a). If we assume all the edges in this graph have 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 trick can be played on the graph formed by the image domain grid (Fig. 1(b) or 1(c)). For example, as illustrated in Fig.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 Fig. 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 Fig. 4. These barriers and the points A, B, C, and D are all chosen automatically with some preliminary processing of the CT slice. Note that barrier induced SP snakes can also be computed in the user interactive setting.

Fig. 4: Barrier induced SP snake on a CT spine slice.

The results in Fig 3 and Fig 4 might seem impressive to many. However if you belong to the category of hard-to-impress type, you must be wondering at this point about the limitations of SP snakes. You certainly have the right concern. One severe limitation of the SP paradigm is inherent in the cost function (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 needed a smoother object contour could it be incorporated in the SP snake? Unfortunately, with the current state of the affairs the honest answer is no. 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.

**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 section 4.1 from a continuous energy minimization perspective. In this section we will illustrate two well known discrete optimization approaches, both based on graph algorithms.

Once again we will view the image domain as a connected graph, *i.e.*, a collection of connected vertices as shown in Fig. 1(b) or 1(c). 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 Fig. 5 we illustrate a cut of the 4-connected image domain grid of Fig 1(b). Note that Fig. 5 contains two graphs— one with hollow pixels (say, background) and the other one with solid pixels (say, foreground). These two graphs are the results of a cut, *i.e.*, removal of a set $R$ of edges from Fig. 1(b). The hollow and the solid pixels are merely symbolic and used to denote the binary labeling resulted from this graph cut.
Fig. 5: A cut of a 4-connected image grid.

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

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

(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).

**Binary Labeling with Minimum Cut**

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

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

(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 models 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 (3) are sometimes referred to as a Markov random field (MRF) model [KZ04].

The mincut (\(i.e.,\) minimum cut) method can be applied to obtain a globally minimum solution for (3) provided the function \(g_{(i,j)}\) obeys a regularity condition [KZ04]:
\[
g_{(i,j)}(0,0) + g_{(i,j)}(1,1) \leq g_{(i,j)}(1,0) + g_{(i,j)}(0,1), \quad \text{for all neighbors } (i, j) \in \Omega.
\]

The trick for minimizing (3) via graph cut is to first represent the cost function (3) as the cost of a cut of a suitable graph. Then obtain the minimum cut by efficient graph algorithms. The graph construction is the basis of this representation that we describe next.

---

**Fig 6:** (a) A source-sink graph. (b) A cut of the graph in (a).

In what follows is a graph construction procedure adopted from [GPS89]. We build a graph \(G\) with \(N+2\) vertices of which \(N\) vertices denote the \(N\) pixels, and the rest two vertices are special ones. One of them is called a source vertex \(s\) and the other one is called a destination vertex \(t\). Among the \(N\) pixel-vertices there are edges between each neighboring pairs (4-neighbors or 8-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 Fig 6(a), where a tiny 4 pixel 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 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 Fig. 6(b). The aim here is to set the edge weights in the source-sink graph \( G \) in such a way that the energy function (3) is represented by the cost of a cut of \( G \).

To represent the first component of the energy function (3) in within the cost of a cut of \( G \) it is written as:

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

\[
= \sum_i p_i \max(0, f_i(1) - f_i(0)) + \sum_i (1 - p_i) \max(0, f_i(0) - f_i(1))
\]

\[
+ \sum_i [f_i(0) - \max(0, f_i(0) - f_i(1))].
\]  

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 (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 (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 (3), \( \sum_{(i,j) \in \Omega} g_{(i,j)}(p_i, p_j) \) 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)

From (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 summation 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 \([g_{(i,j)}(1,1) + g_{(i,j)}(0,1) - g_{(i,j)}(1,0) - g_{(i,j)}(0,0)] > 0\), with weight \(0.5[g_{(i,j)}(1,1) + g_{(i,j)}(0,1) - g_{(i,j)}(1,0) - g_{(i,j)}(0,0)]\); else if \([g_{(i,j)}(1,1) + g_{(i,j)}(0,1) - g_{(i,j)}(1,0) - g_{(i,j)}(0,0)] < 0\), we add an edge from \(j\) to \(t\) with weight \(0.5[g_{(i,j)}(0,0) + g_{(i,j)}(1,0) - g_{(i,j)}(0,1) - g_{(i,j)}(1,1)]\). If any 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 (5) as it is essentially a constant. For a slightly different graph construction see also [KZ04]. Thus a cut of $G$ now represents the energy function (3) up to an additive constant. Having constructed the source-sink graph $G$, one obtains the minimum cut, which minimizes the energy function (3) and provides the corresponding binary labeling of the image.

It is now time to illustrate the mincut on an image shown in Fig 7(a). Note the cells in Fig. 7(a) has 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 the parameters of these models needs to be estimated (say, by maximum likelihood estimation) from training images. Now we are in a position to define the function $f$ in (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 (3) as follows:

$$g_{(i,j)}(0,0) = g_{(i,j)}(1,1) = 0, \quad \text{and} \quad g_{(i,j)}(1,0) = g_{(i,j)}(0,1) = \lambda,$$

where $\lambda$ is a positive number, essentially a tuning parameter that controls the balance between the two summation terms in (3). With such a model we segmented the image in 7(a) by minimum cut to produce the result shown in 7(b).
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. 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 \( I \):

\[
W(i, j) = \exp\left(-\frac{|I(i) - I(j)|^2}{2\sigma_i^2}\right)\exp\left(-\frac{|X(i) - X(j)|^2}{2\sigma_k^2}\right),
\]

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 in (6) are scaling parameters that the user supply. Note that in defining the matrix affinity \( W \), \( i \) and \( j \) pixels need not be 4- or 8-neighbors. The matrix
can be defined for every pixel pair. The affinity matrix $W$ actually defines a graph for the image domain where every pixel connected with every other pixel and the edge weights given by the elements of $W$. Such a graph is called fully connected.

Ncut is a grouping/clustering technique where the clustering can typically be performed either in a hierarchical manner, or in a direct multi-clustering way. In the multi-clustering paradigm, one assigns $N$ pixels to $K$ classes or clusters all in one shot. A special case when $K = 2$ is called 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 mincut 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 rest in the other set. This result is not surprising because all mincut does is minimize 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 (3). The answer to this apparent contradiction is that (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 questions: what other meaningful criterion is there to obtain a somewhat balanced cut? Computationally how efficient are these cuts? Normalized cut or Ncut is such a criterion that 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:

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

(7)

where ‘assoc’ is defined as:

\[
\text{assoc}(V_1, V) = \sum_{(i, j) \in E \cup R} W(i, j),
\]

\[
\text{assoc}(V_2, V) = \sum_{(i, j) \in E \cup R} W(i, j).
\]

Thus ‘assoc’ is the association cost of a vertex set $V_1$ (or $V_2$) with the original vertex set $V$. Note that function ‘cost’ is already defined via (2). This Ncut cost (7) normalizes a cut cost with respect to the association cost. In fact it is easy to see that each ratio in (7) will never exceed 1, thus the Ncut value lies between 0 and 2. Normalized cut method minimizes the Ncut cost (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 $\text{cost}(V_1, V_2)$, while $\text{assoc}(V_2, V)$ will be much larger than $\text{cost}(V_1, V_2)$, 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 (7).
With some algebraic manipulations to the equation (7) Shi and Malik showed that minimizing (7) is equivalent to finding out the eigenvector corresponding to the second smallest eigenvalue of the generalized Eigen system [SM00]:

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

(8)

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 (8) provides a minimum of (7).

In Fig. 8(a) we show a blood cell image that we want to segment (binary label) with Ncut. The affinity matrix we consider for this problem is:

\[W(i,j) = \begin{cases} \exp\left(-\frac{|I(i) - I(j)|^2}{2\sigma_i^2}\right)\exp\left(-\frac{|X(i) - X(j)|^2}{2\sigma_x^2}\right), & \text{for } |X(i) - X(j)| \leq 25, \\ 0, & \text{otherwise.} \end{cases}\]  

(9)

Note that \(W\) is (9) is a sparse matrix, whereas \(W\) in (6) is a dense one. Creating a sparse matrix helps speed up the computation of the eigenvector in (8). Fig. 8(b) shows the second smallest eigenvector \(p\) of the system (8) after we transform the column vector \(p\) to a 2D matrix by the column-major order. Notice that minimization (7) should lead us to a binary labeling, however the matrix in Fig. 8(b) is not a binary one. So, what happened? It turns out that minimizing (7) is approximately equivalent to finding out the second smallest eigenvector of (8), because there is no guarantee that this eigenvector of (8) will be a binary 0-1 vector. There is a very good computational reason behind this approximation. The reason being that finding a binary solution of (7) is an NP-hard
problem [SM00]. For those who are unfamiliar about the notion of NP-hardness, it suffices for now to say that finding efficient algorithms for these problems has not yet been possible, even with many decades of computing science research. Many computer scientists actually believe that efficient algorithms do not exist for these problems. For those readers who really want to know about NP-hardness we refer them to the classical text by Garey and Johnson [GJ79].

Well, that is the good old teaching of computer science. However, we still face the practical question: how do we get the binary labeling from Fig. 8(b)? The engineering solution to this question is to classify the image in Fig. 8(b) into two classes. This is what we have done to produce the binary image in Fig. 8(c). We have used Otsu’s threshold method [Otsu] for this purpose.

Let us close this topic by showing that Ncut can in fact produce multiple labeling. For example, we can perform a ternary labeling of the blood cell image in Fig 8(a) by Ncut method. In this case along with the second smallest eigenvector we also need the third smallest one (shown in Fig. 8(d)). Next, we need a clustering method such as $k$-means [HTF01] to produce the ternary labeling shown in Fig. 8(e). The input to the $k$-means algorithm is both the eigenvectors shown in Fig. 8(b) and 8(d).
Fig. 8: (a) A blood cell image. (b) Eigenvector corresponding to second smallest eigenvalue. (c) Binary labeling. (d) Eigenvector corresponding to third smallest eigenvalue. (e) Ternary labeling.

References


