Chapter 7

Segmentation

The goal of segmentation is to partition an image into disjoint regions such that each region contains pixels that look similar to one another. Segmentation can be viewed as a bottom-up process that groups pixels based on their low-level, local properties such as color or texture. Generally, the goal is to group pixels that are projections of the same object in the scene, although it must be kept in mind that the definition of an “object” is not entirely task independent.

7.1 The image as a graph

For the purpose of segmentation, it is often helpful to view the image as a graph. Each vertex (or node) in the graph represents a pixel in the image, and an edge connects any two pixels that are adjacent in the image. Generally, we shall confine ourselves to 4-adjacency for simplicity, but keep in mind that other types of adjacency may give more complete results in some applications. The edge weights capture the dissimilarity between pixels, often depending upon differences of intensity, color, and/or texture. When larger neighborhoods for adjacency are used, it is common for the weights to also include the image distance between pixels. An example is shown in Figure 7.1.

As we saw earlier with connected components, the goal of segmentation is to find edges in the graph that, when removed, result in subgraphs such that each subgraph contains the pixels for one particular object. These edges are known as a cut in the graph, and because we want to find an arbitrary number of such objects, the division of the image is known as a multiway cut.

Image segmentation can be viewed as a specific application of data clustering, which is a general problem that arises in many empirical domains. One popular approach to data clustering is the hierarchical clustering scheme (HCS). An HCS operates on a fully connected graph containing a vertex for each data point. The weights of the edges are, not surprisingly, the distances (in some space) between

![Figure 7.1: Left: A simple 4 × 3 gray-level image, with intensity values shown. Right: A graph representation of the image, with a vertex for each pixel. In this case, the 4-neighborhood is used to determine the edges, and the absolute difference in intensity is used for the edge weights.](image.png)
7.1. THE IMAGE AS A GRAPH

Figure 7.2: Left: An example of five data points labeled a through e, viewed as a graph with the edge weights indicating the distances (in some feature space) between the points. Middle columns: Initially considering each data point as a separate cluster, sequential iterations of the HCS procedure merge the two closest clusters until all clusters have been merged. Because the weights satisfy the ultrametric inequality, no updating of the weights is needed. Right: The dendrogram is a way to visualize the resulting hierarchical clustering, with the original data points along the horizontal axis and the distances used for merging along the vertical axis.

The data points. The procedure begins by assigning each data point to its own cluster. Then the procedure iteratively applies the following two steps. First, the two closest clusters are merged. Secondly, the weights from the remaining clusters to the new cluster are updated. This simple procedure successively merges clusters until all clusters have been merged.

Figure 7.2 shows a simple example of the HCS that results from applying this procedure to a set of five data points. Since the minimum distance is between vertices a and b, these two are merged first. Then, the distance between vertex c and d is minimum, so these are merged next. Of the remaining distances, the minimum is between e and the cluster \{c,d\}. Finally, the clusters \{a,b\} and \{c,d,e\} are merged. The result is a hierarchical clustering scheme that can be viewed as a dendrogram, which looks like a binary tree with a leaf node for each vertex in the original graph. Dendrograms are usually drawn so that the distance encountered during the merge of each pair of clusters is shown.

You may be wondering why only one step of the HCS procedure was used in this example when we said there were two steps involved. The reason we did not have to update the weights is that the weights in this particular graph satisfy what is known as the ultrametric inequality. Let \( A \) and \( B \) be the two clusters in the graph that will be merged in a particular iteration of the procedure, and let \( \psi(A,B) \) denote the distance between them. Then, for any other cluster \( C \) in the graph, the distance between \( C \) and the new cluster \( \{A,B\} \) needs to be defined once \( A \) and \( B \) are merged so that the procedure can continue. It turns out that if \( \psi(A,C) = \psi(B,C) \), as in the simple example above, the choice is obvious, namely, \( \psi(\{A,B\},C) = \psi(A,C) = \psi(B,C) \). This will happen when the distance function satisfies the ultrametric inequality, namely,

\[
\psi(A,B) \leq \max(\psi(A,C), \psi(B,C)) \quad \text{for all } A, B, \text{ and } C. \tag{7.1}
\]

Recall that a distance function is a metric if it satisfies the metric properties of non-negativity, symmetry, and the triangle inequality. A distance function is an ultrametric if it satisfies not only these three properties but also the ultrametric inequality. It is easy to see that the ultrametric inequality implies the triangle inequality, but not vice versa. It is also easy to see that if three distances satisfy the ultrametric inequality, then two of the values are equal, and the third is no greater than those two. For example, the set of distances \{3, 4, 4\} satisfy the ultrametric property, but the set \{3, 4, 5\} does not. Note that with \{3, 4, 4\}, the two clusters separated by a distance of 3 will be merged, so that \( \psi(A,C) = \psi(B,C) = 4 \).

In the same way, all the weights the example graph above satisfy the ultrametric inequality, because, in each case the distance from all vertices in the new cluster to all other clusters is the same. For example, when merging \{c,d\} and e, the distance between \{c,d\} and \{a,b\} is the same as the distance between e and \{a,b\}, making it easy to determine the distance between \{c,d,e\} and \{a,b\}.

In real-world situations, however, the distance function will not be an ultrametric, and therefore

Copyright (c) 2011 Stanley T. Birchfield. All rights reserved.
we must decide how to define the distance between two clusters \( A \) and \( B \). The three most common approaches are as follows:

\[
\psi(A, B) = \min\{\psi(a, b), a \in A, b \in B\} \quad \text{single-link clustering (7.2)}
\]

\[
\psi(A, B) = \max\{\psi(a, b), a \in A, b \in B\} \quad \text{complete-link clustering (7.3)}
\]

\[
\psi(A, B) = \frac{1}{|A||B|} \sum_{a \in A, b \in B} \psi(a, b) \quad \text{group-average clustering (7.4)}
\]

Figure 7.3 shows a more realistic example of an HCS, using both single-link and complete-link clustering. Note that in the case of single-link clustering, the edges that are used for merging form a minimum spanning tree for the graph. The importance of this observation will be clear in a moment, but let us first consider one of the simplest (yet still effective) methods for image segmentation.

### 7.2 Region growing

A simple and effective technique for image segmentation is to initialize each pixel as a separate region, and then to successively merge adjacent regions that look similar. This is known as agglomerative clustering, or in the context of image segmentation, region growing. The floodfill algorithm that we saw in Chapter 2 is the simplest example of this approach because it merges pixels only when they have identical values. Here we generalize that basic idea to merge pixels when they have similar values. Let us first present the process for growing a single region.

**Copyright (c) 2011 Stanley T. Birchfield. All rights reserved.**
7.2. REGION GROWING

GROW_SINGLE_REGION\((I, O, p, label)\)
1 model.initialize(\(I(p)\) )
2 frontier.push\((p)\)
3 \(O(p) \leftarrow label\)
4 \(\textbf{while NOT} \ frontier.\text{isEmpty}() \ \textbf{do}\)
5 \(p \leftarrow \text{frontier.\text{pop}()}\)
6 \(\textbf{for} \ q \in N(p) \ \textbf{do}\)
7 \(\textbf{if} \ \text{model.\text{isSimilar}()} \ (I(q)) \ \textbf{then} \ frontier.\text{push}(q)\)
8 \(\text{model.\text{update}}(I(q))\)
9 \(O(q) \leftarrow label\)
10 \(\)

The reader should immediately recognize the similarity of this procedure and FLOODFILLSEPARE-OUT. Just as with floodfill, region growing starts with an initial seed pixel \(p\). A stack called a frontier is used to keep track of the pixels on the boundary of the region being grown. At each iteration, a pixel is popped off the frontier, and its neighbors are examined. For each neighbor, if the appearance is similar to the model of the region, that neighbor is added to the region. The process continues until all neighbors do not match the model, at which point the frontier becomes empty.

Three lines distinguish the procedure from the basic floodfill: Line 1 initializes the model, Line 7 compares whether the pixel is similar to the model (rather than identical to the seed pixel), and Line 10 updates the model. The implementation of these lines depends upon the model chosen. One simple, natural, and effective choice is the Gaussian model. By maintaining the mean and variance of the pixel values in the region, we can measure the similarity of a pixel to the model by evaluating the Gaussian at that value. More specifically, for grayscale images we have

\[
\mu = \frac{1}{n} \sum_{i=1}^{n} I(p_i) \tag{7.6}
\]

\[
\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (I(p_i) - \mu)^2, \tag{7.7}
\]

where \(p_1, p_2, \ldots, p_n\) are the pixels in the region. The dissimilarity of a pixel \(q\) is then measured by its distance from the mean, weighted by the standard deviation: \(d(q; I, \mu, \sigma) = |I(q) - \mu|/\sigma\). The reader may notice that this is simply the Mahalanobis distance in one dimension. If we let \(\tau\) be a constant threshold, the function \(\text{model.\text{isSimilar}}(I(q))\) returns true if \(d(q; I, \mu, \sigma) < \tau\) and false otherwise. One nice property of the Gaussian is that there is a natural and intuitive way to select the threshold. Usually \(2.0 \leq \tau \leq 3.0\), where \(\tau = 2.0\) captures \(\pm 2\sigma\) of the Gaussian, or 95% of the area under the curve, while \(\tau = 3.0\) captures \(\pm 3\sigma\) of the Gaussian, or 99.7% of the area under the curve. Another advantageous property of the Gaussian is that it is easy to update the mean and variance by simply maintaining the running sum of values \(s_n = \sum_{i=1}^{n} I(p_i)\) and running sum of squares \(\tilde{s}_n = \sum_{i=1}^{n} I^2(p_i)\). It is easy to show that \(\mu = s_n/n\) and \(\sigma^2 = \tilde{s}_n/n - \mu^2\).

More specifically, the model contains three values: \(s, \tilde{s},\) and \(n\), in addition to the threshold \(\tau\). The model procedures therefore are as follows, where we have taken liberty to rearrange some of the equations to avoid having to compute square root. Note that \text{model.\text{isSimilar}} will always set \(d'\) to 0 whenever \(n = 1\), because the variance is impossible to estimate from a single data value; and the variance estimate is not accurate for small sets. As a result, this simple code below should be slightly modified to allow for merging to take place when \(n\) is small.

MODEL\(:\text{initialize}(\text{val})\)
1 \(\text{model.\_s} \leftarrow \text{val}\)
2 \(\text{model.\_\_s} \leftarrow \text{val} * \text{val}\)
3 \(\text{model.\_n} \leftarrow 1\)
4 \(\text{model.\_\_s}^2 \leftarrow 2.5 * 2.5 \quad ; \text{example threshold to capture} \pm 2.5\sigma\)

Copyright (c) 2011 Stanley T. Birchfield. All rights reserved.
CHAPTER 7. SEGMENTATION

Figure 7.4: Left: An RGB image. Right: Regions found by region growing.

MODEL:isSimilar(val)
1 $\mu \leftarrow \text{model}.s / \text{model}.n$
2 $\sigma^2 \leftarrow \text{model}.\tilde{s} / \text{model}.n - \text{model}.\mu \ast \text{model}.\mu$
3 $d' \leftarrow (\text{val} - \text{model}.\mu) \ast (\text{val} - \text{model}.\mu)$
4 return $d' \leq \text{model}.\tau^2 \ast \sigma^2$

MODEL:update(val)
1 $\text{model}.s \leftarrow \text{model}.s + \text{val}$
2 $\text{model}.\tilde{s} \leftarrow \text{model.} \tilde{s} + \text{val} \ast \text{val}$
3 $\text{model}.n \leftarrow \text{model}.n + 1$

It is easy to extend this basic procedure to color images, or any other image with a vector of values per pixel. For vector-valued images, we simply compute the mean vector by stacking the means of the different channels, and we compute the covariance matrix similarly; the similarity function uses the Mahalanobis distance with the multidimensional Gaussian. Oftentimes the full covariance matrix is not needed, and good results can be achieved by only computing the values along the diagonal (the variances), which saves on computation.

The above procedure grows a single region. Typically we would like to assign a label to every pixel in the image, as in connected components. This involves growing multiple regions. Just as we showed in Chapter 2 that the connected components can be computed by repeatedly applying floodfill, so we can label every pixel by repeatedly applying the region growing procedure. The result is as follows:

REGIONGrow(I)
1 label $\leftarrow 0$
2 for $(x, y) \in I$ do
3 $L(x, y) \leftarrow \text{UNLABELED}$
4 while $L(x, y) = \text{UNLABELED}$ for some $(x, y)$ do
5 $p \leftarrow \text{GetSeedPixel}(I, L)$
6 $L \leftarrow \text{GrowSingleRegion}(I, L, p, label)$
7 label $\leftarrow$ label +1
8 return $L$

The procedure GetSeedPixel returns a pixel such that $L(p) = \text{UNLABELED}$. This can be performed by selecting a pixel at random, or by looking at the intensity values to make a more informed decision.

Despite its simplicity, region growing is an effective technique in many domains. Figure 7.4 shows the results of region growing with a Gaussian RGB model for each region on an example image. It is recommended that, as was done here, a minimum region size is enforced to avoid the effects of noise.

It is not difficult to see that the region growing code above uses an approach similar to group-average clustering, where one cluster is the region being grown, and the second cluster consists of the single pixel being added to the region. The use of 4-neighbor connectedness, rather than a fully

Copyright (c) 2011 Stanley T. Birchfield. All rights reserved.
connected graph, causes subtle differences between this approach and the HCS procedure introduced earlier, but these differences are not important. It would be easy to modify the code to use single-link clustering, where the distance between two non-adjacent pixels is defined as infinity, but complete-link clustering is cost prohibitive for most applications because it requires the pixel to be compared with every pixel in the region. Thankfully, the best results are usually achieved using either single-link or group-average clustering, the former allowing an arbitrary drift in appearance within regions while the later enforces data-driven bounds on the compactness of the similarity of any given region.

7.3 Minimum-spanning-tree segmentation

While region growing is an effective technique, it leaves several important questions unanswered, such as

- What merge criterion should be used?
- How to select the starting pixels?
- Among the several pixels adjacent to the region, which should be considered next?

The merge criterion, which defines what we mean when we say pixels “look similar”, involves three aspects: the value used (e.g., grayscale, color, texture vector, stereo, motion, and so forth); the distance metric used (e.g., Euclidean or Manhattan); and the clustering type (single-link vs. complete-link vs. group-average). Such decisions are largely application specific, and there is not much we can do to answer them at the algorithmic level. However, the other two questions are important, because the outcome of the algorithm will depend heavily on the choice of starting pixels and the selection of the adjacent pixel to consider next. Moreover, a serious drawback of the region growing technique is that only one region is grown at a time, which introduces a bias toward earlier over later regions.

We now describe a simple, elegant algorithm that solves these problems. The method, known as Kruskal’s algorithm, is designed to find the minimum spanning tree (MST) of a graph. Recall that an MST is a set of edges that connects all the vertices in the graph with two additional properties: 1) the set contains no cycles (hence it is a tree), and 2) among all the trees that connect all the vertices, it is the one with the minimum weight, meaning that the sum of the weights of its edges is minimum. The mechanics of Kruskal’s algorithm are simple. First the edges are sorted in non-decreasing order according to their weight. Then the edges are considered one at a time, in order, starting with the smallest-weight edge. For each edge, if the two vertices on either end of the edge are in different regions, then merge the two regions. The procedure continues until all the vertices are in the same region. Why does this approach work? Because, in order to achieve a spanning tree, we know that two vertices will eventually have to merge, and since the edges are already sorted, we can do no better than to merge them using the current edge. Kruskal’s algorithm is therefore a greedy algorithm (meaning it makes decisions locally) that is nevertheless optimal. Such algorithms are rare, so we should be thankful that our problem lends itself to this efficient greedy approach.

The pseudocode for Kruskal’s MST algorithm is as follows:

\[
\text{KruskalMST}(I)
\]

1. \( T \leftarrow \emptyset \)
2. \( \text{disjoint-set.initialize}(\text{width} \times \text{height}) \)
3. \( E \leftarrow \text{constructEdges}(I) \)
4. \( \langle e_1, \ldots, e_n \rangle \leftarrow \text{sortAscendingByWeight}(E) \)
5. \( \text{for} \ (u, v) \rightarrow e_1 \text{ to } e_n \text{ do} \)
6. \( \quad \text{if} \ \text{disjoint-set.findSet}(u) \neq \text{disjoint-set.findSet}(v) \text{ then} \)
7. \( \quad \quad T \leftarrow T \cup \{(u, v)\} \)
8. \( \quad \quad \text{disjoint-set.merge}(u, v) \)
9. \( \text{return } T \)

Copyright (c) 2011 Stanley T. Birchfield. All rights reserved.
Note that the implementation is very similar to the connected components algorithm that we saw before. Both algorithms use a disjoint set data structure, although we called it an equivalence table back then. The auxiliary procedures are as follows: \textsc{Initialize} places each node into a separate set, \textsc{FindSet} returns the “root node” of a set, and \textsc{Merge} merges two sets. These latter two are exactly the same as the \textsc{GetEquivalentLabel} and \textsc{SetEquivalence} procedures of connected components. (In fact, we may want to rename these to be the same, so that we can avoid duplicating code.) The function \textsc{ConstructEdges} determines both the edges and their weights, such as using 4-neighbor connectedness and comparing pixels using the Euclidean distance in color space.

\begin{verbatim}
DisjointSet:Initialize(n)
1  for i ← 0 to n − 1 do
2    root[i] ← i

DisjointSet:FindSet(u)
1  if u = FindSet(u) then
2    return u
3  else
4    root[u] ← FindSet(root[u])
5    return root[u]

DisjointSet:Merge(u,v)
1  a ← min(FindSet(u), FindSet(v))
2  b ← max(FindSet(u), FindSet(v))
3  root[b] ← a

As it stands, Kruskal’s algorithm has a major drawback in that it eventually merges all the pixels into a single region. In fact, Kruskal’s algorithm finds the exact same hierarchical clustering found by our HCS procedure with single-link clustering. But this is not exactly what we want. Rather, our goal is to find disjoint regions in the image. Happily, we can achieve this goal by a simple modification to Kruskal’s algorithm: Instead of merging automatically whenever two pixels are in different regions, we merge only if the pixels are in different regions \textit{and} they look similar. What is so fascinating about this approach is that it essentially performs region growing, but it does so without intentionally selecting a region and growing it. Instead, it simply merges regions in a greedy manner, and when it is done, the regions that remain are the regions that it finds. This modified algorithm does not find a minimum spanning tree, but rather a forest of MSTs, one tree per region. In reality, though, we do not care about MSTs at all, so we will not bother to keep track of the variable $T$ anymore.

\begin{verbatim}
MST-Segmentation(I)
1  disjoint-set.Initialize(width * height)
2  E ← ConstructEdges(I)
3  (e_1, \ldots, e_n) ← SortAscendingByWeight(E)
4  for (u,v) ← e_1 to e_n do
5    u' ← disjoint-set.FindSet(u)
6    v' ← disjoint-set.FindSet(v)
7    if u' \neq v' and disjoint-set.IsSimilar2(w(u,v), u', v') then
8      disjoint-set.Merge(u, v, w(u,v))
9  for (x,y) ∈ I do
10     L(x,y) ← disjoint-set.FindSet(x,y)
11  return L
\end{verbatim}

For comparing an edge to a region, there are several possibilities. The original Felzenszwalb-Huttenlocher algorithm does the following:

\begin{verbatim}
Copyright (c) 2011 Stanley T. Birchfield. All rights reserved.
\end{verbatim}
DisjointSet::IsSimilar2(w, u, v)
1 \textbf{return } w < \min(\max\text{-edge-weight}[u] + k/\text{num-pixels}[u], \max\text{-edge-weight}[v] + k/\text{num-pixels}[v])

where \(k\) is a scale parameter, \(\max\text{-edge-weight}[u]\) is the maximum edge weight in the region \(u\), \(\text{num-pixels}[u]\) is the number of pixels in region \(u\), and similarly for \(v\). Note that, for this function to work, the division must be floating-point division: otherwise, once the region is bigger than \(k\), then no more merging will occur because \(k/N_u < 1\) which truncates to zero, and \(k\) effectively becomes the maximum region size. Also note that the image must be smoothed first in order to convert the pixel values to floating point and to introduce slight differences between identical neighboring pixels, or else the results will be terrible.

The details of the other procedures are just slightly different than before:

DisjointSet::Initialize(n)
1 \textbf{for } i ← 0 \textbf{ to } n - 1 \textbf{ do}
2 \hspace{1em} root[i] ← i
3 \hspace{1em} max\text{-edge-weight}[i] ← 0
4 \hspace{1em} num\text{-pixels}[i] ← 1

DisjointSet::Merge(u, v, w)
1 \hspace{1em} a ← \min(\text{FindSet}(u), \text{FindSet}(v))
2 \hspace{1em} b ← \max(\text{FindSet}(u), \text{FindSet}(v))
3 \hspace{1em} root[b] ← a
4 \hspace{1em} max\text{-edge-weight}[a] ← \max(w, \max\text{-edge-weight}[a], \max\text{-edge-weight}[b])
5 \hspace{1em} num\text{-pixels}[a] ← num\text{-pixels}[a] + num\text{-pixels}[b]

Note that when merging, the weight \(w\) will always be greater than any max weight considered so far (because of ordering), and therefore the \(\max\) in line 4 is unnecessary. Note also that \(\max\text{-edge-weight}\) and \(\text{num-pixels}\) are updated in the root index, and the root index is used in IsSimilar2.

This algorithm is sometimes known as graph-based segmentation, but the name is not very descriptive because other algorithms use graphs, too. Instead, we prefer the name minimum-spanning-tree segmentation. Note that the IsSimilar2 function above uses a new clustering scheme, which we will call smallest-neighbor clustering. It is very similar to single-link clustering. Like single-link clustering, this choice allows for large amounts of drift in appearance within a single region, and therefore it tends to undersegment images. We can modify the MST algorithm to use a different clustering choice.

7.4 Watershed

An alternate approach to image segmentation is the watershed algorithm. In this algorithm, a segmentation function (usually the gradient magnitude) is first computed from the image. Borrowing terminology from geology, the segmentation function is viewed as the surface of a terrain, i.e., a topographical surface. Water falling onto such a terrain flows downhill to a local minimum, and a watershed ridge line is a place where the falling water may flow to multiple minima. A continental divide, for example, is a watershed ridge line.

In the most common version of the watershed algorithm, we imagine puncturing a hole in the segmentation function at each local minimum and submerging the function in water. The water then fills up the catchment basins, which are the regions between the watershed ridge lines. This also leads to the concept of a geodesic influence zone, i.e., the locus of non-labeled pixels that are contiguous with a catchment basin that are closer to that catchment basin than any other. The algorithm is as follows: First we compute a histogram-like data structure which indexes the list of pixels for any given value in the segmentation function. This is a preprocessing step that improves efficiency significantly. This is
the sorting step. Then, for each value, we examine all pixels with that value and either assign the pixel to the closest catchment basin or declare a new catchment basin.

The algorithm is as follows:

**Watershed**($I$)

; initialization
1 $f \leftarrow \text{GradientMagnitude}(I)$
2 for each value $k \leftarrow 0$ to $n_{\text{grad}} - 1$ do
3 $\text{pixellist}[k].\text{CLEAR}()$
4 for each pixel $p \in f$ do
5 $\text{pixellist}[f(p)].\text{PushBack}(p)$ ; precompute pixel lists
6 $L(p) \leftarrow \text{UNLABELED}$ ; all pixels are initially unlabeled
7 $\text{next-label} \leftarrow 0$
8 $\text{frontier.clear}()$ ; flood topological surface one value at a time
9 for each value $k \leftarrow 0$ to $n_{\text{grad}} - 1$ do
10 for each pixel $p$ in $\text{pixellist}[k]$ do
11 if there exists a neighbor $q$ of $p$ such that $L(q) \neq \text{UNLABELED}$
12 then $L(p) \leftarrow L(q)$ ; (in an existing catchment basin)
13 $\text{frontier.PushBack}(p)$ ; continue to grow existing basins one pixel thick each iteration by expanding frontier
14 while $\text{not frontier.EMPTY()}$ do
15 $p \leftarrow \text{frontier.PopFront}()$
16 if there exists a neighbor $q$ of $p$ such that $f(q) \leq k$ and $L(q) == \text{UNLABELED}$
17 then $L(q) \leftarrow L(p)$ ; (unlabeled)
18 $\text{frontier.PushBack}(q)$
19 for each pixel $p$ in $\text{pixellist}[k]$ do
20 if $L(p) == \text{UNLABELED}$ ; (still unlabeled)
21 then floodfill region containing $p$, assigning label $\text{next-label}$
22 $\text{next-label} \leftarrow \text{next-label} + 1$
23 return $L$

In this code, $I$ is the original image, $f$ is the gradient magnitude of the image, and $p$ and $q$ are pixels. We assume that the gradient magnitude is quantized into $n_{\text{grad}}$ bins, where typically $n_{\text{grad}} = 256$ (We used $L$ to refer to this number in a previous chapter, but here we use $L$ to refer to the labeled image). For any gradient magnitude value $k$, $\text{pixellist}[k]$ is an array containing the coordinates of all pixels $p$ such that $f(p) = k$. (In Lines 11 and 16, if more than one neighbor satisfies the test, then one of them is selected arbitrarily.) Then in the for loop, the frontier is populated with all the pixels with the particular value that are adjacent to an existing catchment basin. Note that while growing the catchment basins the algorithm pops elements from the front of the frontier and pushes elements onto the back. The frontier is therefore a FIFO queue, which is important to ensure that all basins grow at the same rate. In the final step, new catchment basins are created whenever a pixel has the current value but has not yet been labeled. The results are stored in a 2D array $label$ that is the same size as $f$ and $I$.

For a robust implementation, the test $\text{AND} f(q) < k$ should be appended to line 11. This ensures that the pixels added to the frontier are adjacent to a pixel that was labeled before entering the for loop; otherwise a contiguous set of pixels with a value of $k$ could all be added, because the next line changes the label of the pixel $p$ that could be the neighbor $q$ of a future iteration. In practice this test is not very important, because it will only affect the result in the pathological case where there exists a
high plateau between two catchment basins. If it is a low plateau, then bleeding to more than one-pixel thick is not a problem at all; in fact, it saves computation later.

Some readers may notice that the algorithm given above is a simplified version of what is usually presented. Typically when the watershed algorithm is described, it involves the building of dams between the catchment basins. That is, when two catchment basins grow to the point that they touch each other, a dam is built between them to prevent water from flowing into the other catchment basin. By simply checking whether a pixel has been labeled, we have avoided the need for building dams and thereby simplified the algorithm considerably. Note, however, that our algorithm does not find the boundaries between regions explicitly; it only labels the pixels. If boundaries are desired, they can be estimated by differentiating the resulting label image. In the worst case, which arises when the true boundary pixel is equidistant from two catchment basins, this approach introduces a one-pixel error in the location of the boundary. For example, if the pixels along a row are labeled 11X22, where the first two pixels belong to region 1 and the last two pixels belong to region 2, the middle pixel will be labeled either 1 or 2 arbitrarily, leading to either 11122 or 11222 as the two possible labelings. Differentiating the first labeling (as an example) yields either 00B00 or 000B0, where the boundary pixel B is misplaced in the latter choice. Introducing dams can alleviate such errors.

The basic watershed algorithm, with or without dams, does not perform very well because it tends to oversegment the image. In fact, the amount of oversegmentation is so great that the results for a typical image are generally unusable. The solution to this problem is to use markers. In marker-based watershed, a preprocessing step determines where to place markers in the image, then during processing a new region is allowed only where there is a marker. This can be accomplished by a single change to the code above: To the test in Line 20 is added the additional test to check whether \( \text{marker}(p) \) is true. Note also that markers are the reason that the test in Line 16 is \( \leq k \) rather than = \( k \). In the basic watershed approach, the value of pixel \( q \) is guaranteed to be greater than or equal to \( k \), because all pixels whose value is less than \( k \) are labeled in Line 21. Only with marker-based watershed is there the possibility of an unlabeled neighboring pixel with a value less than the current value. To summarize, the watershed algorithm is useful only when it is possible to determine beforehand the number and approximate locations of the regions, represented as the markers.

How to determine the markers is application specific. However, when there are multiple foreground objects on a background, it is important to create a marker for the background, in addition to creating a marker for each foreground object. One way to do this is to compute the chamfer distance on an image containing blobs for the foreground objects, then run standard watershed to find the chamfer ridges between the objects. When ORing the markers for the objects and chamfer ridges, we must be sure not to let them touch and bleed into each other. This can be avoided by checking the 8-neighbors of the ridges and removing each ridge pixel that has an 8-neighbor that is a marker. Not guaranteed, perhaps, to work in principle, but works pretty well in practice. Another way is to have two separate marker images and to floodfill them separately when creating a new region. I.e., the marker image is no longer binary but has three values: no marker, foreground marker, background marker. This is clean, works well, and seems fairly fail-safe.