Chapter 14

II - Segmentation

Segmentation is an important and difficult step in many vision applications. Before tasks such as inspection or recognition can be carried out, one wants to know which are the different physical entities (e.g. objects) in the image. One would like to know which pixels belong together. A delineation of regions should be made corresponding to object outlines or the boundaries of certain object surface patches. Methods are rather ad hoc, as is our definition of segmentation altogether.

Segmentation is a chicken-and-egg problem in a sense. How is one to decide which pixels form an entity without knowing what one is looking at? Yet, in order to find out, a reasonable segmentation must have preceeded recognition. Although the idea of applying Gestalt-like ideas of grouping is hanging around for quite a while now, it is not clear how to transform these rather vague rules into an algorithm, neither how to avoid combinatoric explosions in computation time.

The image enhancement and edge detection techniques discussed in chapter ?? are important preprocessing steps that facilitate the actual segmentation. But also other cues, such as velocity and range differences between regions, may be called upon to assist in the segmentation of a scene.

14.1 Thresholding

14.1.1 Thresholding techniques

The simplest to segment are images where the different segments (e.g. objects) have clearly distinct intensities with respect to each other. In such cases simple *thresholding* can be appropriate. Thresholding amounts to somehow determining a critical intensity level and assigning pixels with intensities on either side of this level to an "objects" and "background" class resp. The diversity in thresholding techniques mainly lies in the different selection procedures for that critical level or *threshold*. This very diversity echoes the vulnerability of thresholding.

A first technique applies when the histogram is clearly bimodal, i.e. has two well-



Figure 14.1: *Detail of workpiece image.*

separated peaks. In general, observing that a histogram is bimodal and knowing that care has been taken to give the scene high intensity contrast between objects and background, it is reasonable then to assume that one peak is due to the object(s) and the other to the background. In that case the threshold can be chosen at the intensity corresponding to the minimum between the peaks in the histogram. Consider fig. 14.1. Its histogram is shown in fig. 14.2. Although contrast in this image is not particularly high, we see that the histogram is bimodal, one peak comprising contributions from the background and the other from the object. If we select 35 as the intensity threshold (35 corresponds to the dip between the peaks in the histogram) we obtain the binary image of fig. 14.3.

Another strategy, useful when the relative area of object(s) and background is known (e.g. when workpieces are passing underneath a camera one by one), is to select an appropriate fraction of pixels as object. E.g. knowing that the object covers about 40 % of the image and is brighter than the background, we could select the brightest pixels that together make up 40 % of total image area as "object".

Gradient magnitudes can also provide us with useful directives. Techniques have been proposed that maximize the sum of the gradient magnitudes for the pixel with threshold intensity, the rationale being that such threshold will give an accurate delineation between object and background. Another approach, similar in spirit, is to consider a histogram that only contains intensities from points with high outputs for a Laplacian filter, thus restricting the analysis to points lying close, but not on the edges. Rejecting pixels with low values one will discard both pixels on the edge and the large bulk of pixels well inside the object boundaries or the background. It is hoped the histogram dip (in between the peaks) will be precisely localized in this way. A caveat with this method is that taking pixels close to the edge could rather introduce problems with minimum detection, because pixels are selected precisely from regions where intensities are getting atypical for the region they belong to and the histogram peaks could therefore be ill-positioned and shallow. It can

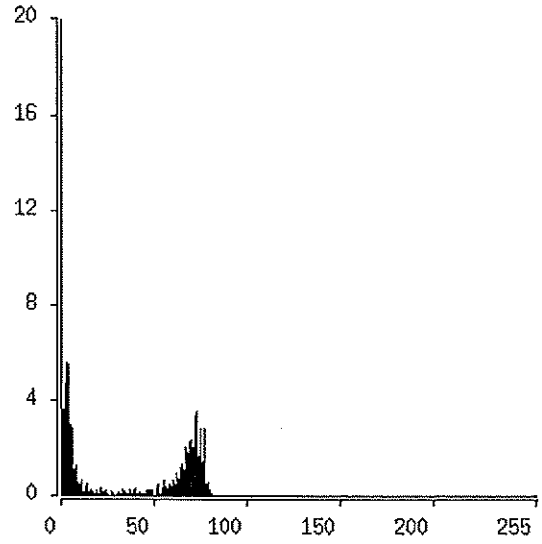

Figure 14.2: *Histogram of fig. 14.1.*



Figure 14.3: *Thresholded output for fig. 14.1.*

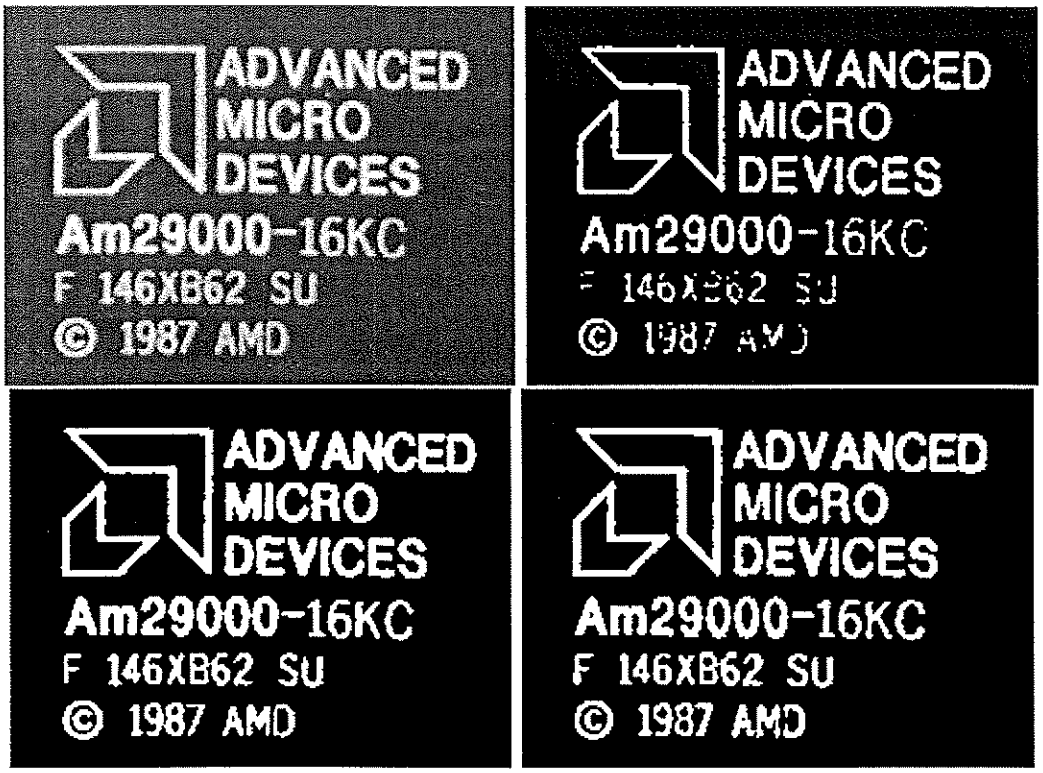

Figure 14.4: *Top left: original image of chip; top right: thresholded image using a fixed threshold; bottom left: result with image specific threshold; bottom right: result using local thresholds.*

also be helpful to use a histogram taking only low gradient magnitude pixels into account, aiming at well-separated peaks. In the same vein, *Otsu's criterion* selects the threshold that divides the image into two maximally homogeneous pixel populations. Each choice for the threshold separates the histogram into two populations of pixels, each with their overall probability p_1 and p_2 and variance σ_1^2 and σ_2^2. The threshold that minimizes the weighted variance $p_1\sigma_1^2 + p_2\sigma_2^2$ is selected. Figure 14.4 shows an example obtained by using this criterion. The goal was to inspect the type number of chips on PCBs. The original image is shown as the upper left part. Selecting a general, fixed threshold for a whole group of such images (i.e. applying the criterion to the average histogram for these images) yields the result in the upper right corner. The bottom row shows the criterion applied to the image taken on its own (left) and applied to 10×10 subimages (right). Although using such locally variable thresholds is often useful, it is seen to add not much to the result obtained with the image-dependent, global threshold in this case. Using the fixed threshold would certainly not suffice in this application, however.

It will not come as a surprise that threshold techniques in practice are usually more the result of trial-and-error for a specific application, rather than the result of some theoretical analysis. Although extremely important in industrial applications of computer vision, the applicability of thresholding must not be overestimated. Its main role in current machine

vision systems is providing a tremendous reduction in bandwith. Once an image has been thresholded, problems like edge detection and the measurement of simple shape characteristics becomes almost trivial and real-time hardware exists to comply with the short cycle times required. Generally thresholding will not provide a satisfactory segmentation of images. Techniques applying several threshold levels (and yielding images with e.g. 3 or 4 instead of 2 output gray levels) won't add sufficient performance. We could think of Lloyd-Max quantization (cf. chapter 5) as an example of such technique.

14.1.2 Enhancement of binary images

Even in high-contrast cases, some pixels can fall on the wrong side of the threshold. When such pixels occur isolated or in small groups, such errors can be corrected by a combination of *erosion and dilation.* These are rank order operations, a class of operations that in turn constitute a subclass of the wider group of morphological operations. Although morphological techniques have become quite popular recently, we will limit our discussion to but a few, traditional examples. Suppose we define a neighbourhood or mask, which will be shifted over the image as with convolution masks. After an operation based on the underlying intensities, the resulting value will be assigned to a pixel in a fixed position relative to the mask. We'll refer to this pixel as the "target pixel". An important difference with convolution will be the non-linear character of the operation, as we will now explain. Suppose there are N pixels underneath our mask. Then we rank order the intensities i of these pixels and obtain

$$i_1 \leq i_2 \leq \cdots \leq i_N.$$

Based on this rank order, we then define our operation as replacing the target pixel's intensity by a value $i_t = f_t(i_1, \cdots, i_N)$. The following three examples of the function f_t correspond to well-known image operations:
1) $i_t = i_1$
2) $i_t = i_N$
3) $i_t = i_{(N+1)/2}$
We recognize the third choice as the median filter (cf. chapter ??). The other two choices form the basis of erosion and dilation.

Erosion and dilation in the case of binary images work as follows. We have the neighbourhood defined by our mask, which for these operations will typically be symmetrical around the target pixel. Suppose the object pixels all have value 1 and all background pixels have value 0. Then erosion replaces the target's pixel value by 0 as soon as one of the neighbourhood pixels belongs to the background, i.e. if a 0 appears in its neighbourhood. This operation corresponds to example 1 above ($i_t = i_1$). This will have the effect of peeling off the object boundaries and removing small blobs of 1's, i.e. blobs that cannot completely cover the neighbourhood mask. Dilation has the opposite effect. Here the target pixel is considered "object" as soon as at least one of its neighbourhood pixels is "object" and corresponds to choice 2 ($i_t = i_N$). This has the effect of putting an additional "object" layer around the object boundaries. We will illustrate these techniques later. Fig.

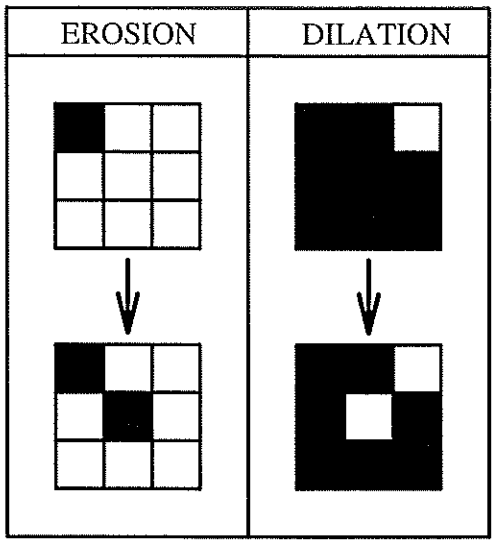

Figure 14.5: *Suppose white is "object" and black "background". The* 3×3 *mask represents a neighbourhood that is taken into account for the operations of erosion (left) and dilation (right). In the example for erosion, the center pixel is made "background" (black) because another pixel in its neighbourhood belongs to the background. Similarly, in the dilation example the center pixel is rendered "object" since one of its neighbours is white.*

14.5 already explains the principle for a 3×3 environment. Of course, modified erosion and dilation schemes can be considered, where the center pixel's value is a more complex function of the neighbourhood values.

Especially the combined effect of dilation and erosion is interesting. Suppose we have an object but due to noise some pixels within the object have been classified as background. Fig. 14.6 shows the image of fig. 14.1 with noise added. Again using the threshold 35 now yields the binary image of fig. 14.7. We see that a lot of noisy spots are present. Combined dilation and erosion will enable us to eliminate these spurious artefacts. One dilation step with a 3×3 neighbourhood brings us at the result shown in fig. 14.8. We see that the spots have disappeared, but the object has also been deformed. A second step, this time an erosion with the same neighbourhood mask, will peel off the false layer around the contours formed earlier by dilation. But because the spurious holes have completely been filled, they remain so and don't reappear. The net effect is that we have corrected the image, as shown in fig. 14.9. Comparison with fig. 14.7 shows that the result is still not completely the same as for the noise-free case. Similarly, erroneous "object" spots on the background could be eliminated through erosion followed by dilation. Notice that the order of the operations is very important: erosion after dilation is not the same as dilation after erosion, even if the same neighbourhood masks are used.

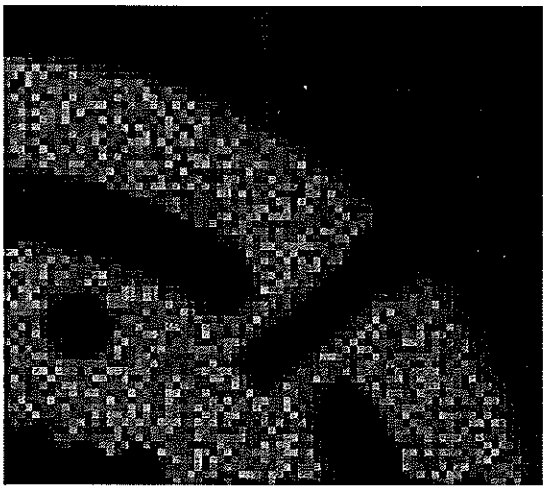

Figure 14.6: *Figure 14.6 with noise added.*



Figure 14.7: *Result of using the same threshold for fig. 14.6 as the one that led to fig. 14.3*
.



Figure 14.8: *The result of one dilation step on fig. 14.7.*



Figure 14.9: *Result of additional erosion on the image of fig. 14.8.*

14.2 Edge detection methods

The thresholding methods were based on the analysis of the image histogram. The underlying assumption was that regions could be identified as peaks in this probability density function of the grey levels. The histogram doesn't contain spatial information, however, and it is clear that we could use such information to our advantage. There are basically two different strategies to detect the segments directly in the image. The first is to extract the boundaries (i.e. "edges") between the segments. The second possibility is to look for regions where some characteristic is more or less constant and to grow these regions until they make contact. In this section we look at the edge approach.

We have already discussed edge detection before. There we tried to identify individual pixels which could belong to an edge. Here we will concern ourselves with the necessary second step: how to combine such pixels into linked lists that describe edges as one-pixel thick, connected lines. The problem with the gradient methods is that simple thresholding of gradient magnitude will result in edges that are several pixels thick at places, whereas they can just as well contain gaps. The fact that the Laplacian (zero-crossing) type of edge detector immediately yields thin, connected (even closed) edges has been one of the reasons for the initial optimism about its usefulness. As we have already seen, these edges all too often lack perceptual relevance, however.

14.2.1 The Hough transform

The Hough transform pure (we won't deal with the Generalized Hough Transform, which basically is a kind of template matching, a technique discussed in chapters ?? and ??) uses the symmetry of shapes to extract them in lower dimensional spaces. To give two simple examples, a straight line remains invariant when shifted along its own direction and a circle doesn't change when rotated about its center.

The principle is best illustrated for the straight line case. Suppose we would like to detect straight lines (e.g. straight edges in object outlines) in all possible positions and orientations. In general, if we want to detect a shape under these conditions, we have to deal with 3 degree of freedom (a 3-parameter group) at least: two components of translation and 1 of rotation. Straight lines however remain invariant under a one-dimensional subgroup of the translations. In fact, the image projection of a straight line is fully characterized by 2 parameters, whatever the transformation group we let act, as long as it preserves collinearity. Notice that we don't take endpoint positions into account. If we write the equation of a straight line as

$$y = ax + b,$$

we see that two real numbers a and b fully characterize the line. On the other hand, fixing a point (x, y), all (a, b) pairs of lines containing the point are found to be related by a linear relationship also:

$$b = (-x)a + y.$$

Thus, to detect straight lines, one can scrutinize all points of interest (e.g. points with a

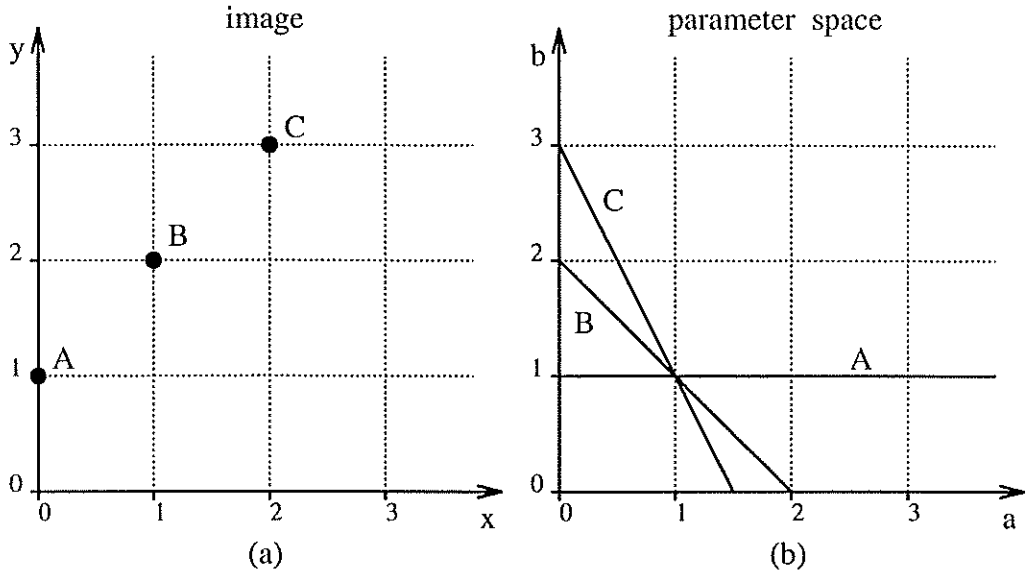

Figure 14.10: *(a) three collinear image points* $A(0,1), B(1,2), C(2,3)$ *lying on the line* $y = x + 1$, *(b) the three corresponding lines in parameter space:* $b = 1$ *for* A, $b = -a + 2$ *for* B, *and* $b = -2a + 3$ *for* C.

high probability of lying on an object outline). For each such point we draw the above line in a (a, b)-parameter space. The situation is sketched in fig. 14.10. Part (a) of the figure shows three collinear point in the image. Part (b) shows the corresponding lines in the (a, b) parameter space. These lines indicate all (a, b) pairs yielding a straight line through one of the points at a time. Where the three lines intersect, the corresponding (a, b) parameter pair corresponds to the straight line going through all three of the image points.

As far as the implementation of the Hough transform is concerned, the parameter space will have to be discretised. A finite number of cells are created. When constructing the straight lines a counter will be incremented at each cell where the lines pass, i.e. the cells receive "votes". In order to detect straight lines one then has to look for cells with high votes, i.e. many lines passing through.

The previous version of the Hough transform has the disadvantage that the (a, b) domain is not bounded. In particular we have problems in dealing with vertical or near-to-vertical lines ($a \to \infty$). This problem can be circumvented by redefining our two parameters as shown in fig. 14.11. In that case, the straight line is expressed as

$$x \cos\theta + y \sin\theta = \rho.$$

This time each point will add a cosine function in the (ρ, θ) parameter space. The values of θ can be restricted to the interval $[0, \pi[$ and the magnitude of ρ never surpasses the largest distance an image point can have to the origin. The image on the left in fig. 14.12 shows the (ρ, θ)-Hough transform for a square. Four peaks clearly stand out, one for each sides of the square. The image on the right in fig. 14.12 shows the Hough transform for a circle.

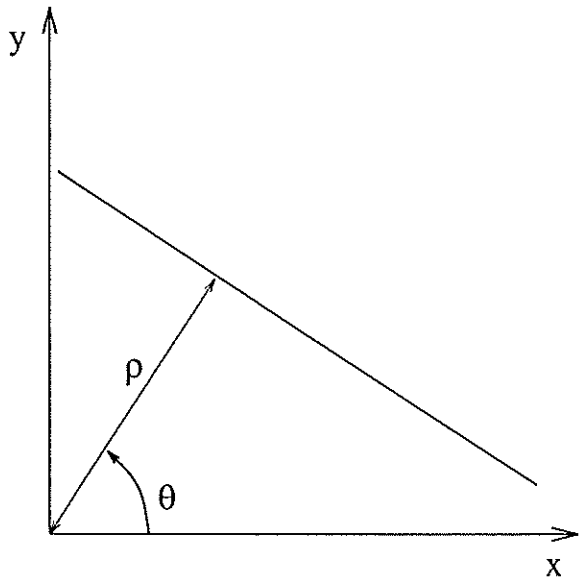

Figure 14.11: *Alternative parameter representation of straight lines on the basis of radial distance ρ and orientation θ. Both parameters will remain finite. In particular, the angular specification of orientation remains in* $[0, 2\pi)$.

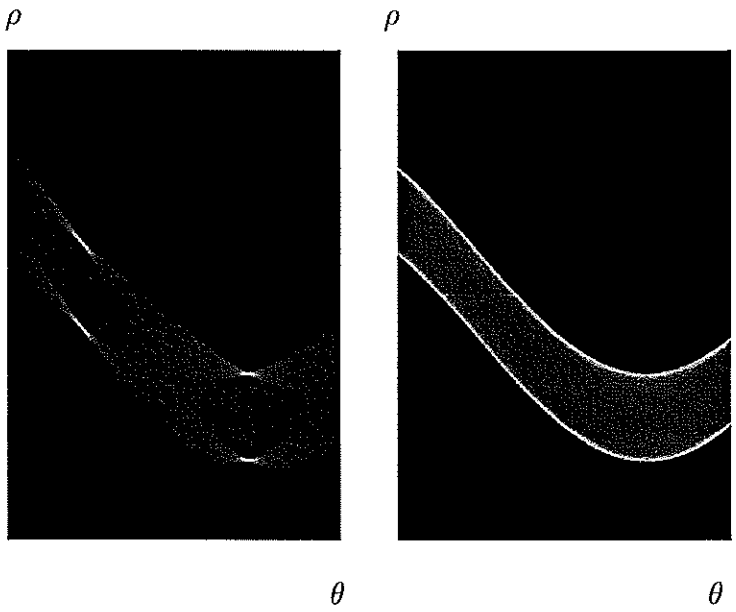

Figure 14.12: *Left: Hough transform of a square, with the sides standing out as four peaks; right: Hough transform for a circle, without salient peaks. In both cases the ordinate corresponds to ρ and spans the interval $[-\rho_{max}, \rho_{max}]$ for the images (see text) and the abscissa corresponds to θ, spanning the interval* $[0, \pi[$.

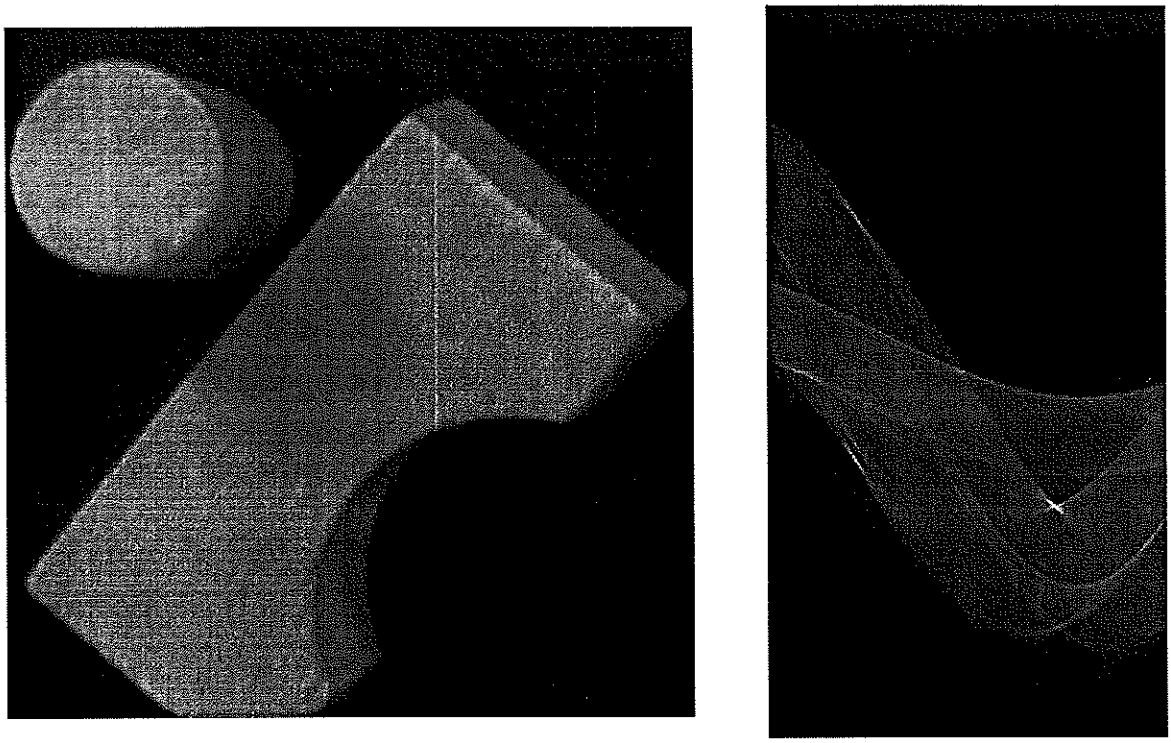

Figure 14.13: *Image and its Hough transform. Several peaks emerge. One is very salient and corresponds to the long straight side of the largest object. The other peaks are much less easily perceived and several points receive high votes without representing any straight segment.*

First, we should emphasize that the intensity of this image has been normalized to cover the whole range. Although the peaks look equally bright in both images, the highest values in the circle's Hough space are substantially lower than for the square. Nevertheless we see that for this shape without any straight edges some points obtained higher votes than others. Just looking for local maxima therefore doesn't guarantee a correct interpretation. A more realistic example is given in fig. 14.13. Several peaks can be observed, one of which is particularly prominent and belongs to the long straight line of the object at the lower right. As can also be observed, other peaks are not very salient and some false peaks will compete with the genuine candidates. The detection of the peaks is a hard problem indeed.

Robustness of peak extraction can be improved by weighting the contributions on the basis of some confidence index. One could e.g. use the intensity gradient at the point (cf. chapter ??), thereby attributing a higher weight to points which are more probable to really belong to edges. This measure will in general reduce false peaks while hardly affecting the genuine peaks.

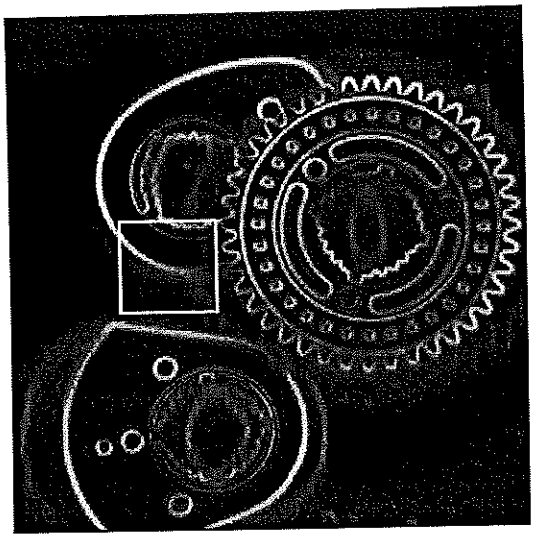

Figure 14.14: *Magnitude of intensity gradient for workpiece image. The highlighted area is illustrated as a 3D plot in fig. 14.15.*

14.2.2 Salient edge point linking

The problem with the Hough transform is that one needs a pretty precise description of the type (i.e. shape) of edges to be expected. We could e.g. detect straight lines, circles, or ellipses. Hough transforms have already been used for all these shapes. The Generalized Hough transform allows the detection of any predefined shape. However, it is clear that such *a priori* knowledge will often be lacking. We need methods that extract edges for arbitrary scenes. In fact, most edge linking strategies are rather *ad hoc* and based on a preference for smooth rather than jigged edges, high intensity gradient values rather than low ones, etc.. We confine our discussion to giving an example.

As already noted, once edge detection masks such as the Sobel pair have been applied (cf. chapter ??), the points with high intensity gradient form ribbons rather than one-pixel thick edges, and at places these ribbons will die out where edge contrast is locally too low. A Sobel gradient magnitude image is shown in fig. 14.14. Discard the highlighted region for a moment. One of the many edge linking methods is then the following. Look for the pixel with the highest intensity gradient. If this value is above some minimal threshold, T_{start}, then start growing the corresponding edge. This is done by considering all neighbouring pixels with an intensity gradient magnitude above some second, floating threshold T. Locally we will find a ribbon, as already explained. Think of this ribbon as a ridge in the landscape (array) of gradient magnitude values, as shown in fig. 14.15 for the window region in fig. 14.14. Moreover, suppose this landscape is flooded and the water surface has height T. We would now like to "walk" along the middle line of the ridge. As we want to keep our feet dry the threshold T (water level) is regulated by some deus-ex-machina (the program) in order to make the ridge a predefined number of pixels wide.

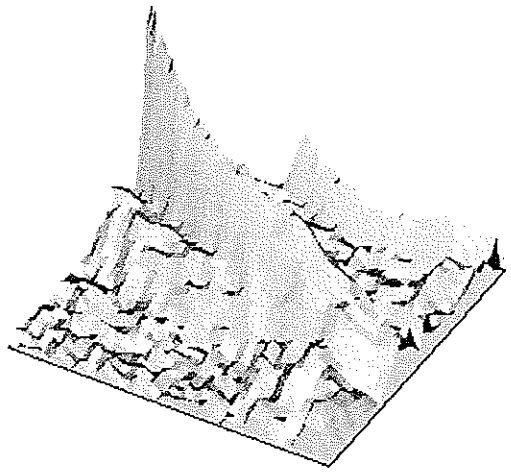

Figure 14.15: *Magnitude of intensity gradient for highlighted area in fig. 14.14 drawn as height above the horizontal image plane.*

Thus, we consider only pixels with a gradient value above a floating threshold, which is regulated so that the pixels considered form a ridge of predefined width. If the ridge is too wide, the threshold T is increased, if the ridge gets too narrow, T is decreased. This keeps the number of pixels to be handled reasonable, while taking care of not "loosing" the ridge by putting the threshold T completely above its top and keeping the middle line of the ridge well aligned with the real edge by only considering the most outspoken edge pixels above a threshold which is not too low either. The floating T cannot take on values below a minimum T_{min} or above a maximum T_{max}, however. Especially the former restriction is important. Otherwise, we could extend edges into areas with so slight a variation in intensity that we would end up with a random walk instead of edge following. Figure 14.16 shows the result for fig. 14.14. Notice that edges are often incomplete, and sometimes erroneously connected. A real-time VLSI implementation of this edge tracking algorithm is available.

14.2.3 * Dynamic path search

A priori knowledge leads to the success of the Hough transform. We can also use less precise information to our advantage, however. We could e.g. know the starting and endpoints of an edge, or at least the regions where these are due to lie. *Dynamic path search algorithms* can then be used to connect them by an optimal path, optimal being defined on the basis of some cost function to be specified by the user. Alternatively, the user could draw a prelimimary, roughly positioned version of the edge interactively. The computer then has to refine the position, again on the basis of a cost function. If the cost function has been chosen appropriately, we will see "crawl" the edge towards its optimal

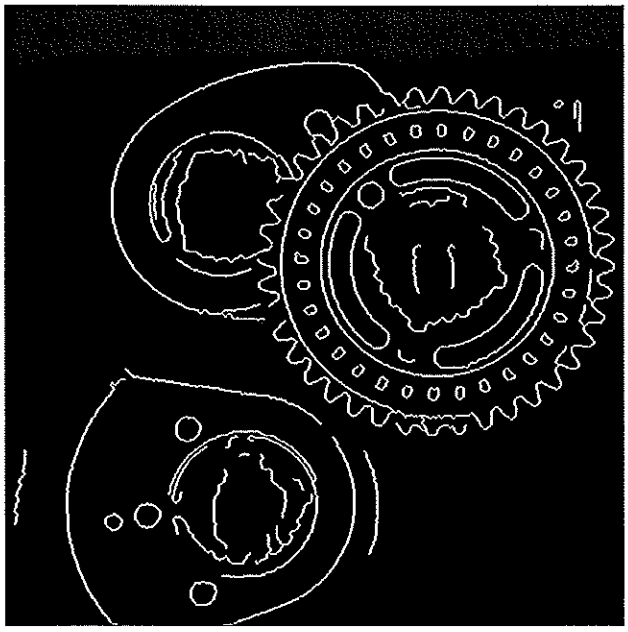

Figure 14.16: *Edges for fig. 14.14 produced by an edge following algorithm.*

position, hence the name *snakes* in that case. Snakes won't be discussed. Let us start with a concise description of the dynamic path search type of algorithm.

Dynamic path search techniques belong to the field of *dynamic programming.* The methods resemble the edge linking strategy, but are more flexible (have e.g. a stronger gap filling capacity) in that they are guided by a cost function that should be supplied by the user. More precisely edges ("paths") are compared on the basis of their cost. Edges with the lowest cost are selected. These methods usually require rather precise *a priori* information on path position (e.g. starting point and end point) and are therefore mainly useful in predictable environments or with interactive help. Delineation of organs or arteries in medical images for diagnosis is a good example. The physician can indicate regions of interest and the computer traces the best outlines, thereby possibly supplying some information on metrical data like areas, widths, lengths, etc. Even if such human help is available, the task of searching the optimal edges is a formidable one. Simply trying all edges satisfying the constraints (e.g. having specified end points) is ruled out. This section investigates strategies that are better suited.

Nomenclature

Before we can introduce dynamic path search, we have to give some nomenclature on *graphs.* A graph consists of *nodes*, which in our context usually represent pixels. The theory presented here has many other applications though, also in vision. Every graph we will consider has a unique node s called the *start node.* In our edge detection context, this

start node will correspond to a given starting point of the edge to be found. Certain pairs of nodes are connected by directed *arcs*, which in general represent operators available to solve the problem. In our case, a pixel will be connected with its 8 (or 4) neighbours and these connections represent directions in which we can grow the edge. If an arc is directed from node n to node n', node n' is said to be a *successor* of n and node n is said to be the *parent* of n'. The process of identifying the successors of a node is called *expansion* of the node. The number of successors emanating from a given node is called the *branching degree* of that node and normally is denoted as b. A *tree* is a graph in which each node (except the start node which is also called *root node* for a tree) has only one parent. We have to deal with graphs that are not trees. Also are the arcs undirected, i.e. the arcs could be used in either direction; such arcs are also referred to as *edges* in graph theory. Often the arcs are assigned *weights* representing a *cost* (or possibly a reward) associated with their inclusion in the final solution (edge). We use the notation $c(n, n')$ to denote the cost of the arc from n to n'. A sequence of nodes $n_1, n_2, \cdots, n_k$, where each n_i is a successor of n_{i-1}, is called a *path* of length k from node n_1 to node n_k. The *path cost* is normally understood to be the sum of the costs of all the arcs along the path. In special cases other costs might be considered such as the maximal arc cost in the path.

The problem then is to find a path, given some boundary conditions, e.g. the endpoints or their approximate position, that minimizes the path cost. Path cost in our case might be a function penalizing changes in edge direction or the selection of arcs connecting pixels with low intensity gradient or directions not orthogonal to the gradient direction. Once the cost function has been selected, we have to devise an algorithm to find the minimum cost path. Several strategies are open to us, ranging from fast but unreliable up to rather time consuming but leading to the real optimum.

Gradient descent

Suppose we have decided on the cost $c(n, n')$ to be added with each pixel that is appended to the path. Also suppose the region or pixel where the path should start is given, as well as the region or pixel where it should end. *Gradient descent* is probably the simplest scheme that can be employed in order to solve this problem. We select a node from the starting region (if this consists of only one pixel the choice is evident). After expanding the node, the successor adding the lowest cost is chosen. Then this successor node is expanded and the operation repeated until we finally arrive at a pixel of the goal region. This process has to be repeated for every candidate starting pixel. The path with the lowest cost for all these pixels is then the solution. Note that in this particular formulation of the problem the name "descent" is not really appropriate since cost will normally increase. The name derives from a widely used strategy for optimization problems, many of which are guided by the minimization of an "energy function", starting from some configuration with high values and evolving towards lower values, continuously moving in the (downwards) direction of the gradient.

Fig. 14.17 shows a possible situation. We want a path minimizing a cost to be specified yet and connecting the starting node on the left with any pixel of the right column of

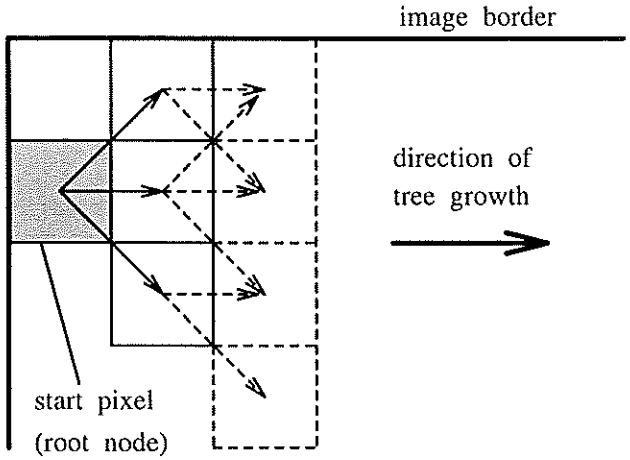

Figure 14.17: *Pixels form a tree (in general a graph) and paths are formed by joining up admissible links between the pixels.*

a user-supplied search window. Starting from the left, a number of pixels on the second column can be reached (in the case of the figure 3), depending on the allowable steps (arcs). With each of the corresponding steps a cost is associated. Gradient descent selects the step with the lowest cost and completely discards the others. Then, starting from the new node (pixel) on the second column again the cheapest of the three possible steps is chosen, bringing one in the third column. This process is carried on until the goal column is reached. We will illustrate the method for two regions of fig. 14.18. This image shows a medical image, a so-called "angiogram". This angiogram is an image of the blood vessels in the brain, obtained by the used of a contrast substance. Figure 14.19 shows one of the windows in detail. The task is to find the cheapest path from a fixed point on the left to any point on the right. The cost function used for all examples based on this image is given by

$$\left(\sum d \, \|\nabla f\|\right)^{-1} \tag{14.1}$$

where the sum is taken over all nodes on the path, ∇f is the intensity gradient and d is 1 for a horizontal and 0.75 for a diagonal step. The role of the latter variable is to penalize longer paths. This function is chosen rather ad hoc, and choosing an appropriate cost function is one of the main problems with dynamic path search algorithms in general. The result is shown in fig. 14.20 and is seen to correspond to an edge.

Gradient descent is a one-way strategy. One never returns to reconsider earlier decisions. Although extremely fast when compared to genuine dynamic path search algorithms, there is no guarantee that the real solution will be found. This stands in sharp contrast with exhaustive search and some techniques to be explained later. For this method to work, the guiding cost function should be sufficiently robust against the influence of noise. Any local mistake can bring the process off-track. But this is not enough. Even if the cost function is evaluated on the basis of noise-free data, there remains the problem of *local optima*. Gradient descent never considers to sacrifice a "cheap step" now, in order to get access to

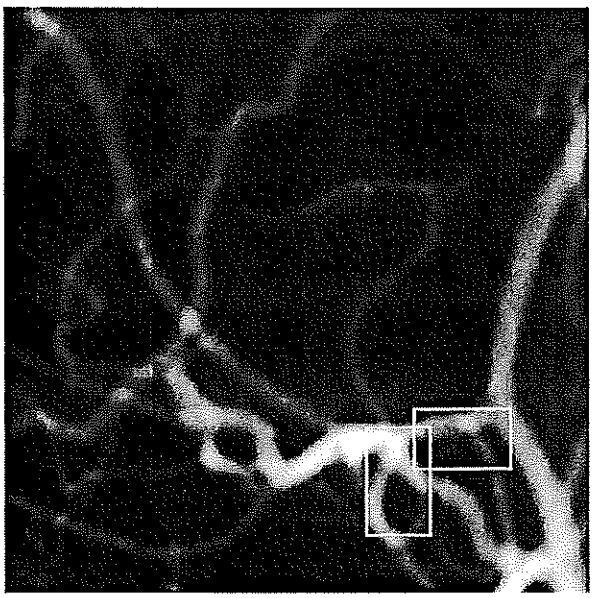

Figure 14.18: *Angiogram image with two excerpts that will be considered further highlighted.*

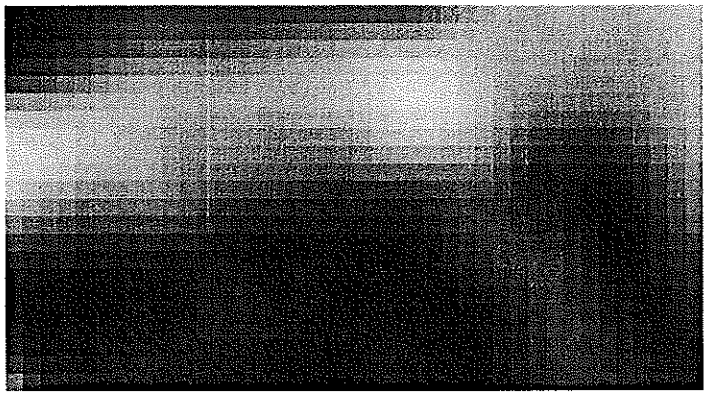

Figure 14.19: *Detail (horizontal window) from fig. 14.18.*

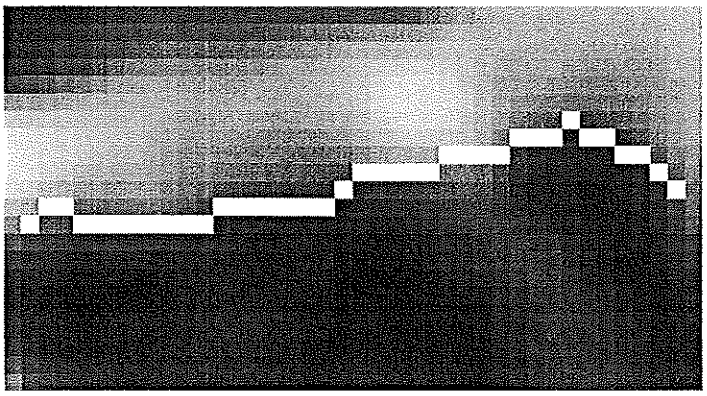

Figure 14.20: *Edge found by gradient descent for fig. 14.19.*

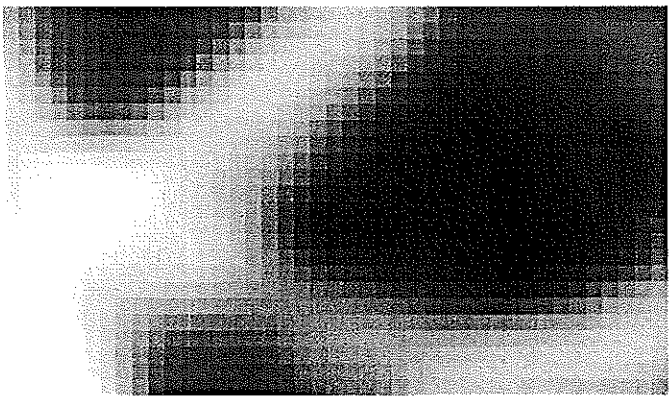

Figure 14.21: *Detail image (vertical window rotated) from fig. 14.18.*

lots of "cheap steps" later. This explains why such algorithms are sometimes referred to as "greedy". The possible negative consequences are illustrated with the second window, shown in detail in fig. 14.21. Using the same gradient descent algorithm and again starting from an appropriate pixel, we find the result of fig. 14.22. This time the result is quite useless. In fact, we wanted to obtain an optimal fit to the upper edge of the lower vessel, but the program lost track at the vessel crossing. The reason is immediately clear: at the crossing high gradients point at the wrong direction.

Gradient descent is found every so often in vision, with a wide array of problems. So even if it is not quite an effective approach to segmentation, the student of computer vision will come across it sooner or later, e.g. again at the end of this chapter. We used the occasion to introduce it in a context where we can also discuss some of the alternatives. In the following section we discuss *path array techniques* that resolve the shortcomings of gradient descent.

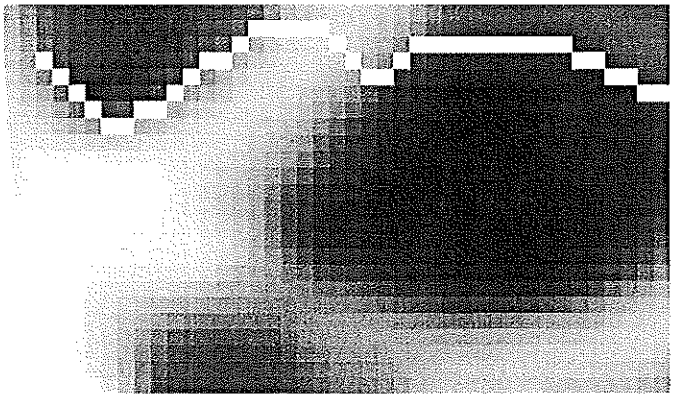

Figure 14.22: *Edge for fig. 14.21 found by gradient descent.*

Path array techniques

A one-pass algorithm We will illustrate the principle on the basis of the examples used for the illustration of gradient descent. Again we first expand the start node, to arrive at three possible pixels in the second column. These nodes are expanded further and all successors on the third column are considered. Here some nodes (pixels) will be successors of more than one node of the second column. The cost associated to the nodes in the third column is the minimum cost to reach them. The cost of such a path could e.g. be calculated as the sum of the costs of its individual steps. For each node of the third row we keep a pointer to the parent on the cheapest path. This process is carried on until the right column of the window (i.e. the set of goal nodes) is reached. Then, in order to find the optimal (cheapest) path connecting the starting node with this column, we look for the node with the lowest cost. This node is connected with the starting node via the cheapest path. The construction of this path requires *backtracking*: the pointers are followed back until the starting node is reached. If the task would have been to connect the starting node with a specific node of the right column, one should backtrack from this node.

If we apply this technique to the windows in fig. 14.18, we obtain the results in figures 14.23 and 14.24. The problems encountered with gradient descent have successfully been avoided.

This search is suboptimal in the sense that paths with costs far above that of the optimal one will completely be developed just the same. Moreover, we have solved several optimization problems instead of solely the one under scrutiny. The algorithm yields an array of cheapest costs incurred to arrive in almost any of the window pixels when starting from the starting node. The array of these costs is sometimes referred to as the *path array*, hence the name of the section. On the other hand, bookkeeping requirements are reduced to a minimum in a case like this one, and this is an advantage over the best-first strategies still to be discussed.

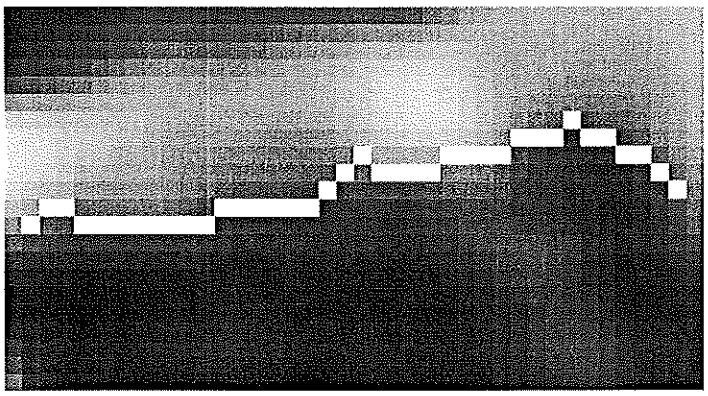

Figure 14.23: *Optimal path for image 14.19.*

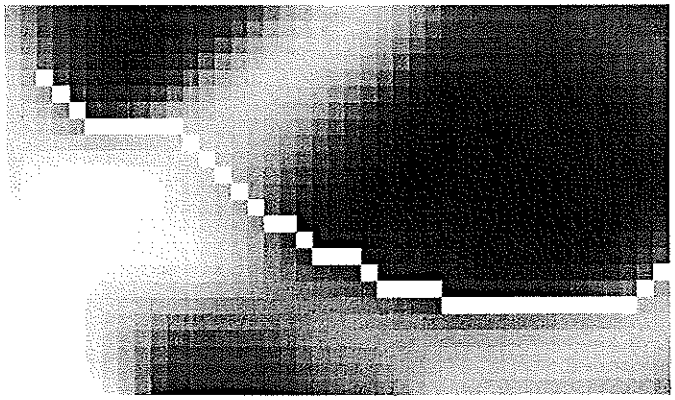

Figure 14.24: *Optimal path for image 14.21.*

The F* Algorithm The search as described above is not generally applicable. It cannot handle paths "meandering" through the image. In particular, the scheme as discussed above doesn't allow a path to turn back and run from right to left for a number of steps. It is clear, however, that this is a rather serious restriction. As an alternative the so-called *F* algorithm* can be considered.

F* iteratively finds an optimal path in an image from a starting pixel (or a set of candidate starting pixels) to a terminating pixel (or a set of candidate terminating pixels). Suppose the minimal cost path from a start pixel s to a goal pixel g is to be found. For this purpose a path array P is considered that this time will be constructed following a multiple pass strategy. When F* stops, each entry p of P will contain the total cost of a minimal cost path from s to p. In the following discussion a pixel p is written as (i, j) and C(i,j,i',j') is the cost for adding the step from pixel (i, j) to pixel (i', j')

The F Flowchart*

1. First, each entry of P is initialized to infinity, except for s which has the initial value equal to $C(s)$, the cost associated to the initial node itself (e.g. $C(s) = 0$).

2. The first iteration involves a *top to bottom, row by row* updating of P. The entries on the i-th row are subject to two adjustments. First they are adjusted from left to right according to rule 14.2 and then from right to left according to rule 14.3

$$P(i,j) := \min \begin{pmatrix} P(i,j), \\ C(i-1,j-1,i,j) + P(i-1,j-1), \\ C(i-1,j,i,j) + P(i-1,j), \\ C(i-1,j+1,i,j) + P(i-1,j+1), \\ C(i,j-1,i,j) + P(i,j-1) \end{pmatrix} \tag{14.2}$$

$$P(i,j) := \min\left(P(i,j), C(i,j+1,i,j) + P(i,j+1)\right) \tag{14.3}$$

   Each node should be assigned a pointer, indicating the neighbour to which the pixel has eventually been linked (the neighbour yielding the lowest cost).

3. Additional iterations alternate between a *bottom to top* pass (with row indexing reversed, the bottom row having $i = 1$), followed by a top-to-bottom pass. In each iteration the updating rules 14.2 and 14.3 are used and pointers are updated.

4. The algorithm stops when in a complete pass no entries in P did change. The optimal path from s to g can be found by backtracking from g until s is reached.

It is interesting to note that if $C(i,j,i',j') = C(i',j')$, i.e. if the cost of adding a pixel to the path is independent of the direction of the corresponding step (arc), then even the simple bookkeeping to update the pointer can be omitted. In fact, the use of pointers can be abandonned completely in that case. Backtracking starts from g as usual and one can then proceed by iteratively moving to the neighbour with the smallest value in P until s is reached.

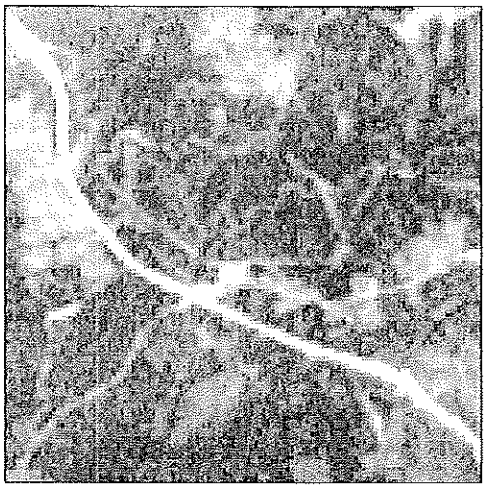

Figure 14.25: *Aerial image.*

We illustrate the F* algorithm for an application in remote sensing. Consider the aerial image in fig. 14.25. The task is to extract the linear structures in the terrain. We assume the user indicates the endpoints. These points will normally be chosen at positions where the linear structures are salient enough so as to make this part of the task easy. The computer should then complete the missing part in between, minimizing a cost. In our example, this cost is of the form $C(i,j) = (255 - F(i,j))^a$ where $F(i,j)$ is the output of a convolution filter that highlights narrow linear structures (i.e. a matched filter for line detection). The F values are rescaled to lie in the interval [0,255] and are shown in fig. 14.26. The result for $a = 1$ is shown in fig. 14.27. Fig. 14.28 shows the output of the same algorithm for $a = 2.4$. As can be seen, the optimal path follows the real road trajectory more closely. This can be explained by the fact that the second cost function penalizes the inclusing of pixels with low filter values more heavily. As a consequence, it becomes increasingly beneficial to make a small detour and follow the road more closely. The resulting path is somewhat longer indeed. Again it is clear that the choice of a good cost function is far from trivial.

F* is guaranteed to find the optimal solution. On the other hand, this algorithm invests quite some effort in finding solutions we are not interested in. Indeed, it can be noted that only the start pixel s is essential to F^*. Each entry in the final path array P contains the minimal cost to reach this entry from s.

Best-first search

The Best-First Algorithm The strategies to be expounded now strike a balance between the two previous ones. On the one hand, they use problem-specific information to guide the path growing process quite selectively, just like gradient descent. On the other hand, they return the optimal solution if applied properly.

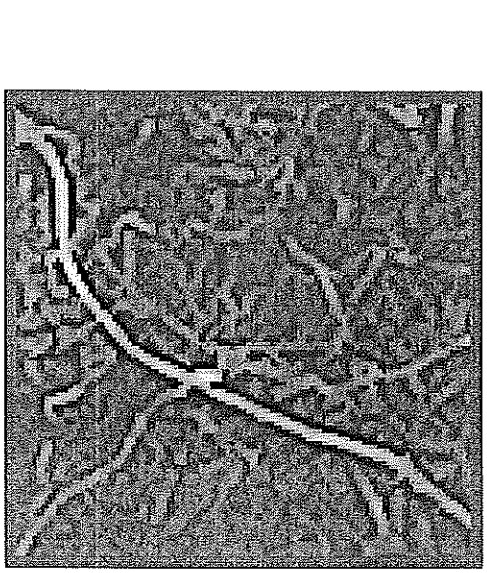

Figure 14.26: *Output of line detection filter for fig. 14.25.*

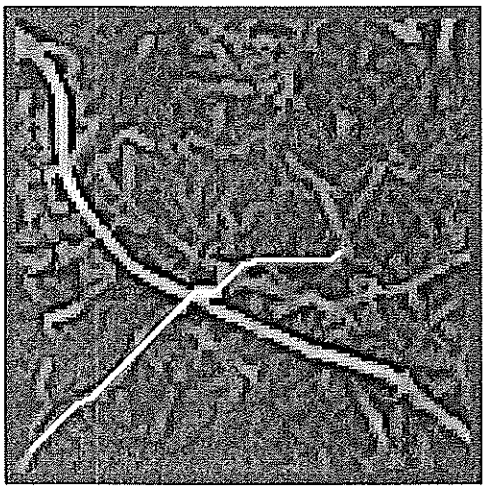

Figure 14.27: *Road found by F^* search with $a = 1$.*

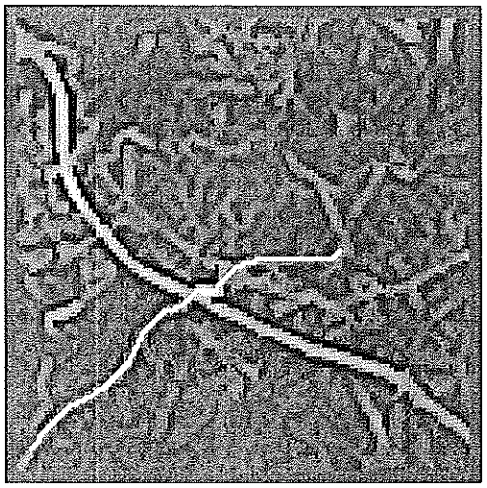

Figure 14.28: *Road found by F* search with* $a = 2.4$.

As before, we consider evaluation functions $f(n)$ that have to be minimised. We consciously avoid the term "cost function" since the evaluation functions may be slightly different than actual path costs, as is to be explained. They will be used to guide the path growth as with gradient descent, but in a more subtle manner. Several versions of such *best-first* or BF strategies are available, but they can all be captured by a single framework. Before we discuss this, we need to define our data structure a bit more precisely. We will keep two dynamic lists of nodes, called OPEN and CLOSED. Nodes that are or have been under consideration but still await expansion are put in the list OPEN, whereas nodes under consideration that already have been expanded are in the list CLOSED.

The BF Flowchart

1. Place the start node s on OPEN.
2. If OPEN is empty, exit with failure.
3. Remove a node n with minimal f from OPEN, and place it on CLOSED.
4. Expand node n, generating all its successors.
5. If any of the successors is a goal node, exit successfully with the solution obtained by tracing the path along the pointers from the goal back to s.
6. For every successor n' of n:
   (a) Calculate $f(n')$.
   (b) If n' was neither on OPEN nor on CLOSED, add it to OPEN, attach a pointer from n' to n, and assign $f(n')$ to n'.

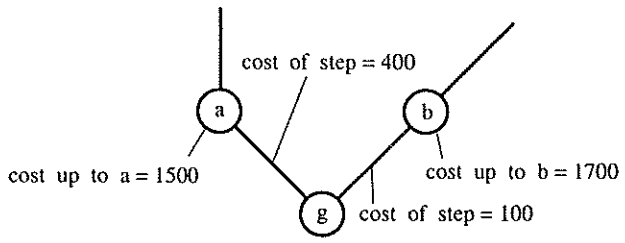

Figure 14.29: *Situation where BF search would stop at the wrong solution (see text).*

(c) If n' already was on OPEN or CLOSED, compare the newly computed $f(n')$ with the previously assigned value. Only if the new value is lower, substitute it for the old value and redirect the pointer accordingly (i.e. as to point from n' to n). If the pointer was redirected and node n' was on CLOSED, move it back to OPEN.

7. Go to step 2.

The attentive reader will have noticed that there is a risk of missing the optimal solution by ending the search too soon. The problem lies at step 5: as soon as one of the successors is a goal node, BF immediately stops, without taking account of the effect associated with the additional cost of the last arc. As an example, consider fig. 14.29. Suppose that the evaluation function is simply the sum of the costs of individual steps included in a path. Suppose two paths have been generated up to the nodes a and b, resp. Both have the goal node g as a successor. BF will expand the path through a first, since it has the lowest cost. With g a successor BF then stops. The total cost of the path is 1900. However, the total path through b has the lower cost of 1800 and the one through a clearly is not the optimal solution. Therefore, often a modified version is used, the so-called BF* algorithm, that is free of this hazard. The principle is very simple: wait until the new arc is part of the evaluation function before accepting the solution. We will not change the algorithm here, but will include the principle in our formulation of the A* algorithm, which we will discuss later on.

In its current form, the algorithm describing BF is merely a strategy sketch. Different choices for f will lead to different best-first algorithms. The ideal evaluation function $f(n)$ is the cost of the cheapest path from s to a goal node and running through the node n. This choice is clearly out of reach. Nevertheless, one can try and approximate this cost by an appropriate choice of f. In the following paragraphs we will study gradual specializations of f, i.e. restrictions on the class of evaluation functions considered. Solution via best-first search is increasingly efficient for these restricted classes, i.e. specializations of best-first search.

The Z and Z* Algorithms Suppose we have a candidate path. Further suppose we have a node n, lying on this path. If C is the cost function we should minimize and $C(n)$ indicates its value for the portion from n to the end of the path, then C is *recursive* if for

every node n in the path

$$C(n) = F(E(n), C(n'))$$

where n' is the immediate successor of n along the path and $E(n)$ stands for a set of local properties at the node n, e.g. a cost $c(n, n')$ assigned to the arc connecting n and n'. It is then possible to evaluate the cost of a given path stepwise, starting from the goal node and moving back. F is sometimes referred to as the *rollback function*, hinting at this backwards stepping procedure. Examples of recursive rollback functions are the maximum or the sum of the arc costs along the path, but also the cost function (14.1) introduced earlier. Sums being the most widespread types of cost functions, we may conclude that usually recursiveness applies. Examples of non-recursive cost functions are the difference between the most expensive and the cheapest step costs along the path and

$$C_{nr}(n') = \sqrt{c(n, n') + C_{nr}(n)} \ . \tag{14.4}$$

where (and this is crucial to remember!) $C_{nr}(n)$ is the cost of the path from the starting node s up to node n rather than from n up to the goal node g as for C in the definition of recursiveness. In this case, when given the cost of the path from n' to g, there is no way to know what the path from n to g would cost. The cost function C_{nr} allows forward recursiveness, but not the backward recursiveness as demanded in the definition.

Recursiveness is a most important property if the best-first algorithm is to function efficiently. Suppose some *deus ex machina* supplies us with the cost $C(n)$ of an optimal path from where we are up to a goal node. This information is only useful if it can be used to calculate the cost of the global path, i.e. the path already grown plus that remaining part. Then it would be possible to compare paths and to select the best one. This in not possible with non-recursive rollback functions. Imagine the following situation. One is currently at node n and a path to a goal node is within reach via the node $n+1$ to a goal node $n+2$. With costs to be calculated according to eq. (14.4), the small remaining path from n to g has the cost

$$C(n) = \sqrt{c(n+1, n+2) + \sqrt{c(n, n+1)}} \ .$$

Now, suppose this cost is given and we try to take a step back. $C(n)$ and the cost $c(n-1, n)$ of going from previous node $n-1$ to n do not suffice to calculate the cost

$$C(n-1) = \sqrt{c(n+1, n+2) + \sqrt{c(n, n+1) + \sqrt{c(n-1, n)}}}$$

of going from node $n-1$ to the goal node $n+2$. In contrast to what happens with recursive rollback functions, the precious information about future costs cannot be exploited.

Of course, arrived at a node n, we cannot know what the exact cost $C(n)$ of the remaining optimal path would be. However, this cost could be estimated and the estimation can then be used instead, provided recursivity applies. Suppose an estimate $h(n)$ of the actual cost $C(n)$ of the best remaining path is available. The cost of the total path is

obtained by using $h(n)$ instead of $C(n)$ and then applying the rollback function to compute an estimated cost of the total path, i.e. by calculating the cost recursively, starting at n and going to the root node s. This version of the BF algorithm – i.e. BF with estimated future costs taken into account – is called the Z algorithm. If we further modify the algorithm by delaying the termination test as was done with BF*, Z becomes Z*. Z* is guaranteed to find the optimal solution path if

1. $h(n)$ underestimates the real cost of the remainder of the path,
2. the rollback function is order-preserving for any two cost forecasts $h(n')$ and $h'(n')$, and where n' is assumed to be a successor node of a node n: $h(n') \geq h'(n') \Rightarrow F(E(n), h(n')) \geq F(E(n), h'(n'))$,
3. the rollback function is order-preserving for parents n_1 and n_2 of a common successor n: $F(E(n_1), h(n)) \geq F(E(n_2), h(n)) \Rightarrow F(E(n_1), h'(n)) \geq F(E(n_2), h'(n))$.

We won't give a rigorous proof of this property of the Z* algorithm. The first pair of demands can be easily understood. The first demand states that future costs should be underestimated, the second that this underestimation should result in an underestimation of the overall cost of the path. This is crucial, since it guarantees that suboptimal paths, with an overall cost larger than the optimal one, say C_{opt}, will never be able to block the partially developed optimal path, which will have an underestimated cost smaller than C_{opt}. Indeed, by underestimating the real cost of paths which have not been completed yet, (suboptimal) paths approaching the goal node and therefore approaching their real cost, will be handicapped when compared to the incomplete alternatives.
On the other hand, the third demand should make sure that the algorithm wouldn't discard an alternative but superior path, passing through the same node n but via another parent, because of an accidental choice of the *estimated* future cost $h(n)$ (in fact, the requirement says that the precise value of $h(n)$ should be irrelevant in determining relative values for F). The demand insures that the same decision in reorienting the pointers will be made as when the real future cost would be known exactly.
The above conditions make sure that the cost function lends itself to the useful introduction of the estimate $h(n)$, thereby yielding an effective evaluation function $f(n)$. Again taking the maximum cost and the sum cost as examples, we see that both satisfy demands 2 and 3.

The A* Algorithm If we use for the rollback function a simple sum $C(n) = c(n, n') + C(n')$ with n' the immediate successor of n along the path and $c(n, n')$ the (positive) cost associated to the corresponding arc, we arrive at the A* algorithm. Again, we have to introduce an estimate of the cost for the part of the path not found yet. We then have as the evaluation function for a node $f(n) = g(n) + h(n)$, with $h(n)$ the estimation of the cost for the remainder of the path and $g(n)$ simply the sum of costs of all the arcs already in the path. Notice that also $g(n)$ can be calculated recursively, this time from the starting node,

making this cost function especially efficient to handle. Recursiveness of F as introduced earlier is not enough to achieve this. As an example,

$$F = c(n, n') + \sqrt{h(n')} \ ,$$

with n' a successor of n, is recursive yet requires calculations to recommence for the whole path upon expansion of the end node. Notice that this rollback function differs from the one given by eq. (14.4). In the case of the sum, however, local information is perfectly sufficient to update f. We now give the organisation of the A* algorithm, noting that the Z* algorithm can easily be generalized from it:

The A Flowchart*

1. Place the start node s on OPEN.
2. If OPEN is empty, exit unsuccessfully.
3. Remove from OPEN a node n for which f is minimum. Put it on CLOSED.
4. If n is a goal node, exit successfully with the solution obtained by tracing back the pointers from n to s.
5. Otherwise, expand n, generating all its successors. For every successor n':
   (a) If n' is not already on OPEN or CLOSED, estimate $h(n')$ (an estimate of the cost of the best path from n' to some goal node), and calculate $f(n') = g(n') + h(n')$ where $g(n') = g(n) + c(n, n')$ and $g(s) = 0$. Put n' on OPEN. Set a pointer from n' to n.
   (b) If n' is already on OPEN or CLOSED, direct its pointers along the path yielding the lowest $g(n')$. If n' required pointer adjustment and was found on CLOSED, reopen it.
6. Go to step 2.

Note that instead of checking whether CLOSED is being revisited (step 5), one could leave out this test and always put the newly selected node on OPEN. No time is spent on doing the testing, but duplication of work may result, having several copies of the same nodes on OPEN and therefore expanding the same nodes several times.

Further restricting the class of evaluation functions, it can be made unnecessary to reconsider nodes on CLOSED in step 5 (and typically there are many!). This is the case if

$$h(m) - h(n) \leq c_{min}(m, n)$$

where $c_{min}(m, n)$ is the cost of the cheapest path from node m to node n. This *consistency assumption* imposes a piecewise underestimation property on h and a monotonous increase of the cost attributed to a path when it grows. Upon consistent application of the same,

reasonable estimation rule for h, this property normally holds. This explains its name. That nodes on CLOSED need not be reconsidered when consistency applies, can be proven as follows. Suppose a node m' is already on CLOSED. Is it possible then that there is a node m of which m' is a successor and that leads to a cheaper path to m' than the one available? This new cost to reach m' will be denoted by $g_{new}(m')$ and the cost of the currently selected path by $g_{old}(m')$. Furthermore, $g_{new}(m') = g(m) + c(m, m')$, where $g(m)$ is the cost of the alternative path up to m. Reconsidering the node m' would be necessary if $g_{new}(m') < g_{old}(m')$, or equivalently,

$$g(m) + c(m, m') < g_{old}(m') \ .$$

Adding $h(m')$ on both sides and using $h(m) - h(m') \leq c(m, m')$, it follows that

$$g(m) + h(m) < g_{old}(m') + h(m') \ .$$

This, however, is incompatible with the way A* works. Indeed, on the left hand side is the estimated cost for the optimal path through node m used by the algorithm when it arrived at m. On the right hand side is the estimated cost as used by the algorithm at the time it arrived at m' through the current path. A* would only put m' on CLOSED after it had expanded this node, implying that there would be no cheaper competitors at that stage. The above relation contradicts this, since the alternative path ending at m would certainly look more promising to A* and this alternative path would be explored first. If this alternative path through m would not have reached m by the time m' was further expanded as part of the old path, it would have been even cheaper, also following consistency. Thus, if the new alternative is really competitive it will be explored well on time and, in particular, before the old path is grown beyond m'.

Simplified A* Algorithms For $h(n) = 0$, A* is sometimes referred to as the *uniform-cost* or branch-and-bound algorithm. The consistency assumption clearly holds (all costs are assumed positive). From a step-by-step point of view, this algorithm clearly is more efficient than general A*. Not only is there no estimate $h(n)$ that has to be calculated, moreover we don't need to update information in nodes that are closed. Nevertheless, uniform-cost will typically be slower in providing the optimal solution. Incomplete paths are not penalized for their being incomplete (as is the case with normal versions of A* via $h(n)$) and the growing of the optimal path will therefore be interrupted more often in order to further investigate such incomplete but suboptimal paths.

So-called *breadth-first* algorithms are even a stronger simplification, obtained by putting $h(n) = 0$ and using for $g(n)$ the number of arcs linked up in the path from s up to n, i.e. $c(n, n') = 1$. This algorithm is e.g. useful for the detection of connectedness in binary images, where the existence of a path consisting only of "object" pixels and connecting points in two "object" regions labeled different is of interest. If such a path is found, the regions can be merged. It is important to note, however, that even under such circumstances the elimination of cost forecasts $h(n)$ needs not be a particularly good idea. Although such simplification saves time considering each step separately, overall efficiency

may decrease also with such task. Breadth-first search boils down to expanding the nodes on OPEN in the same order in which they were put on the list.
Depth-first search is a bit the counterpart of breadth-first in that sense. It also uses $h(n) = 0$, but always expands the most recent node on OPEN.
Both depth-first and breadth-first are blind techniques that can only be applied in rather small graphs. Depth-first can easily get lost in some vast, but otherwise irrelevant part of the graph, whereas the optimal path was maybe rather short. Breadth-first will examine a huge number of paths if the optimal path is rather long. The examined paths spread as a kind of wavefront of equal length through the graph, without giving very promising paths a fair chance.

The advantages of best-first search are clear. By introducing the possibility of adding estimated future cost and by using cost information as a means for selective expansion of nodes, we can expect these algorithms to spend less time on needlessly growing inferior paths. Nevertheless, F* for instance can be more appropriate, depending on the application. If the optimal path doesn't clearly stand out, many alternative paths will have to be checked by best-first search also. Moreover, if edges are not strongly meandering as in the remote sensing example, F* will need only few passes to arrive at its final path array. The additional bookkeeping computation costs of A* may under these circumstances prove counterproductive. F* has the additional advantage that changing the goal region is possible without appreciable effort. Only backtracking from different positions is needed.

14.3 Region based methods

The previous methods focused on the transition between the regions to be segmented. As an alternative, the techniques in this section work from the inside of the regions towards their boundaries.

Region-based methods start with the detection of some very "homogeneous" regions of the image, e.g. regions with intensity variations below a certain threshold. These regions are then maximally grown but only to the extent that they remain sufficiently homogeneous. This could e.g. be interpreted as the intensity variance being lower than some second (usually somewhat higher) threshold. Homogeneity could also be defined as constant textural features, e.g. having constant intensity variance in specified neighbourhoods or containing similarly oriented patterns. Region growing stops at pixels that violate the homogeneity criteria if they were included.

Just as with the choice of cost functions for dynamic path search, the choice of appropriate homogeneity criteria is not a straightforward matter. Variance as suggested above is e.g. doomed to fail once regions are becoming large, since adding a pixel will have scarcely any effect on overall variance even if its value deviates substantially from the expected range. It may therefore be preferable to compare values at the new pixel (and possibly its small neighbourhood) and assessing the acceptibility of its individual value when compared to minimum and maximum thresholds (e.g. defined on the basis of the region's mean and variance). An abundance of (admittedly often rather ad hoc) criteria has been proposed in

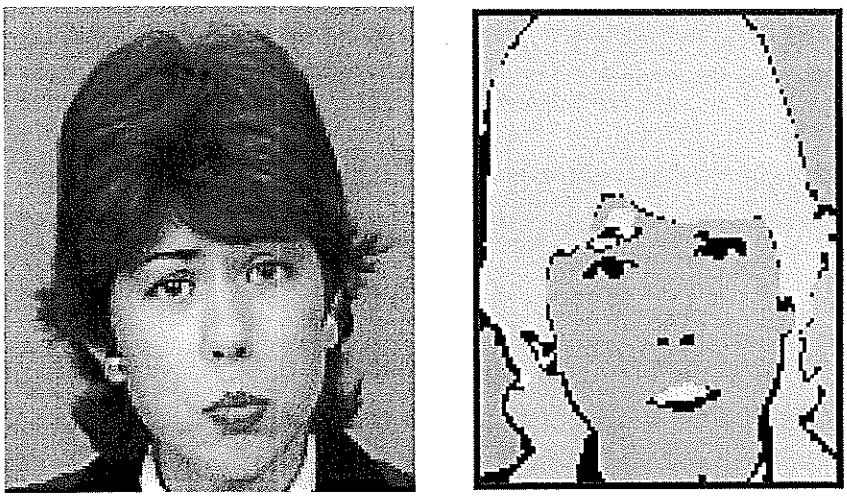

Figure 14.30: *Example image for region growing (left) and the result (right).*

the literature. As an example we illustrate the forementioned strategy for the image of fig. 14.30. First we look for regions of low intensity variance. These regions are then grown by appending all pixels that have an intensity between two thresholds. The thresholds are updated based on small numbers of neighbouring pixels already in the region. The intensity of the candidate new pixel should not be too different. The result is shown on the right, with different regions given different, constant gray levels. It is clear that this homogeneity criterion will often segment incorrectly. In particular, there is a risk of different regions blending together through low-contrast "gaps" in their boundaries.

Region growing is a one-way process. If the initial homogeneous "seed" regions to be grown are chosen erroneously, the method is bound to fail. This might occur when several such "seeds" are initiated in what should eventually become a single region. Seeds might also be put at region boundaries, since locally such regions might also look homogeneous (e.g. when edges are very blurred or contrast is locally low). These problems are solved by considering the merges of similar ánd neighbouring regions and splitting of existing regions into several parts. Such decisions again are taken on the basis of homogeneity considerations. Whereas merging is an operation that is rather trivial, this is not the case for splitting, since usually there are many ways to split a region into subparts. Most algorithms will only allow very specific splitting operations (see e.g. the quadtrees introduced in chapter ??).

14.4 * Hybrid methods

As a last, but important category, we here describe methods that combine enhancement of homogeneity within regions with discontinuity extraction and growth. Suppose we have K points where we know some quantity, e.g. intensity or range. These data are denoted by

d_k. The d_ks may be sparse, i.e. only specified at some locations (e.g. range data obtained from stereo are usually sparse. Our goal is to derive new values $u_{i,j}$ at each position in the image such that discontinuities are detected and the values between the discontinuities are smoothed in order to suppress noise. On the other hand, we of course have to stick to the data to some degree. The reader will have noticed the similarity of this problem with the one we had when solving the optical flow problem (following the Horn & Schunck method of chapter 10). Also there we had to strike a balance between fitting closely to the data and flow field smoothness. We then introduced the principle of regularization, that will be used here also. The important difference is that at the time we didn't care about discontinuities (in the optical flow field), whereas here we outline a method to prevent them being blurred by the smoothness term. The algorithm to be described is coined *the method of weak continuity constraints* and is due to Blake & Zisserman.

A weak continuity constraint is one that is usually obeyed but may be broken on occasion, when there are pressing reasons to do so. E.g. at most positions in an image, intensities correspond to positions on a smooth part of a surface, where the intensities should vary smoothly. On the rim of such surface, however, abrupt changes may well be genuine and should be preserved. At positions where a smoothness constraint is broken, a penalty α is charged. Whether the constraint will be broken is then a matter of weighting this penalty against other costs. If breaking a constraint leads to a total saving in these other costs that exceeds the penalty α, then breaking that constraint is deemed worthwhile. Clearly a large α makes the constraint almost unbreakable. The energy to be minimized could then be chosen as follows:

$$E = \sum_{k=1}^{K}(u_{i(k),j(k)} - d_k)^2 + \sum_{i=1}^{N}\sum_{j=1}^{N}(\begin{array}{l} \lambda^2(u_{i,j} - u_{i-1,j})^2(1 - l_{i,j}) \\ +\lambda^2(u_{i,j} - u_{i,j+1})^2(1 - m_{i,j}) \\ +\alpha(l_{i,j} + m_{i,j})) \end{array}$$

with $i(k), j(k)$ the i, j positions where data d_k are available, $N \times N$ the dimension of the image, and $l_{i,j}$ and $m_{i,j}$ indicating the presence of a discontinuity: $l_{i,j} = 1$ means the presence of a discontinuity, decoupling $u_{i,j}$ from its left neighbour, and $l_{i,j} = 0$ no such discontinuity, similarly $m_{i,j} = 1$ means the presence of a discontinuity decoupling $u_{i,j}$ from its lower boundary, and $m_{i,j} = 0$ no such discontinuity. This is the *membrane model*, because the smoothness term contains (approximations of) first order derivatives, resisting tension but not bending. Introducing a discontinuity releases tension, but a penalty α is introduced instead. This is necessary, because otherwise the optimal solution would clearly be to decouple all the points and to stick to the data exactly. If the second term is replaced by e.g. the square of the Laplacian (second-order derivatives), then we have a *plate model*, resisting bending also. This enables one to detect both step and crease discontinuities. With membrane models only step discontinuities can be detected. The parameter λ is a measure of the elasticity of the string. It can be shown to play the role of a scale parameter, a large λ corresponds to interactions over large distances.

The problem with minimizing E is that it is non-convex. [1] This increases the complexity of optimization dramatically. Before we introduce some methods that can handle the type

[1] A convex functional has a Hessian matrix that is positive definite.

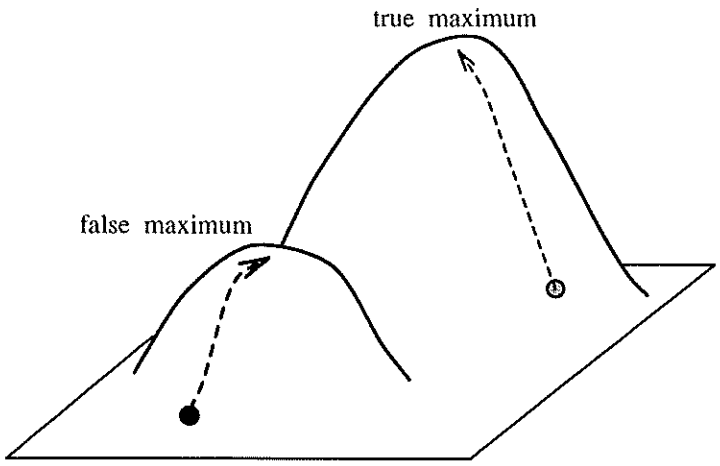

Figure 14.31: *Hill climbing in a two-dimensional seach space. Height gives the figure of merit. The "optimum" reached depends on the starting point, entailing the risk of starting from a point outside the "basin of attraction" of the real optimum.*

of energy functions encountered here, we sketch the problems with non-convexity. Fig. 14.31 gives an intuitive illustration for a low-dimensional search space (two-dimensional in this case). The choice of the variables allows us to move in a horizontal plane, above which we have put out a kind of figure of merit. This function has to be maximized (rather than minimized like the energy considered in the regularisation problem). This doesn't change anything to the problem, but makes the interpretation of the drawing somewhat easier. *Hill climbing* will then consist of continuously changing the two variables such as to realize the steepest increase in the figure of merit. Starting from the point in the front the resulting path doesn't lead us to the highest top, but rather to the second highest position. The method has failed to yield the true optimum. Note also that starting from another point (indicated in the back) could bring one to the correct maximum in this case. Hill-climbing (gradient descent) therefore only guarantees to bring one in a local optimum, not a global optimum. For a convex energy function, optimisation can be implemented using a very simple technique: each variable in turn is changed a little in the direction that improves on the evaluation criterion.

To illustrate the non-convexity of the energy function E, we will perform a kind of Gedanken experiment. Again, it is useful to think about this problem with the help of the mechanical analogue of the membrane. We can imagine a membrane that is fixed to the different measurements d_k by springs. If we forget about introducing discontinuities for a moment, the solution will correspond to the minimal energy state of the membrane. Suppose for instance, we have a bright square on a dark background, both of constant intensity. Initially, we fit a membrane without tears (discontinuities). Instead of having sharp, step-like behaviour at the square boundary, we will find a blurred, sigmoid-like profile. If the contrast between square and boundary is sufficiently high, we know the final minimal energy state will be one of a flat square disconnected (by a dicontinuity) from

a flat background. If we now start with the tearless state and move a point $u_{i,j}$ within the square but close to the boundary upwards in order to push it into the plane of the square (it is pulled down by the tension in the membrane), then this point is brought in closer agreement with the final solution, yet the tension in the membrane will increase and so will the overall energy. This is in contrast with what we should find if we hope to successfully apply gradient descent. To make things even worse, the energy function has discontinuities.

We now describe two methods for finding global optima of non-convex functions. The first – the technique of *simulated annealing* – is very general, time consuming and stochastic. The second – *graduated non-convexity* or GNC – has been developed specifically for the type of functions dealt with here, is computationally much more efficient, and is deterministic.

Simulated annealing is a powerful, general method for finding global optima of functions which have many local optima. It is based on an algorithm for simulating statistical mechanical systems. Instead of direct descent, some randomness is introduced into the descent path. This avoids sticking in local minima. The degree of randomness is controlled by a temperature parameter, which is initially high. As the system approaches the global minimum, it is allowed to cool. The temperature T is reduced from some starting value towards 0. Changes δ_i in each u_i are tested according to a random rule. Similarly, discontinuities are introduced or deleted randomly. In order to avoid the processing wandering around in the energy space indefinitely, changes have to be evaluated somehow. The following rule is applied: if $\Delta E < 0$ then the change is accepted, but if $\Delta E \geq 0$ then it is accepted randomly with probability

$$e^{\frac{-\Delta E}{T}},$$

and otherwise rejected. T is gradually decreased. While T is still large, energy increases are often accepted, enabling the system to jump out of local minima. As T approaches 0, the system "freezes", and only small jumps are still possible. The system becomes more deterministic.

An alternative is offered by the so-called *Graduated Non-Convexity* or GNC approach. In GNC, a one-parameter family of energy functions $E^{(p)}$ is defined. As p decreases from 1 to 0, $E^{(p)}$ is initially a convex approximation to E, and moves steadily towards the real E. The algorithm consists of tracking a local minimum $u^{(p)}$ of $E^{(p)}$, as p is reduced from 1 to 0. In order to illustrate the working principles, we limit the following discussion to the 1D case with dense data, the study of a string so the speak, when sticking to our membrane analogue. The energy then simplifies to

$$E = \sum_{i=1}^{K}(u_i - d_i)^2 + \sum_{i=1}^{N}\left(\lambda^2(u_i - u_{i-1})^2(1 - l_{i,j}) + \alpha\, l_{i,j}\right),$$

with notations as before. The second term shows discontinuous behaviour: given the u_i and writing $t = u_i - u_{i-1}$, we will have a discontinuity wherever $|u_i - u_{i-1}| > \sqrt{(\alpha)}/\lambda$. Denoting this term as $g^{(0)}(t)$, the situation is depicted in figure 14.32. We now replace this

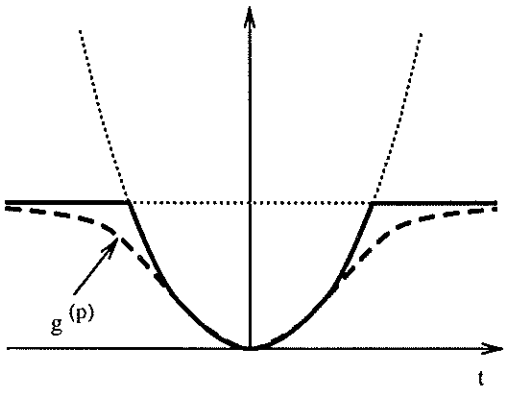

Figure 14.32: *The local interaction energy function $g^{(p)}$ is used to produce the family of cost functions $E^{(p)}$.*

term, i.e. $g^{(0)}(t)$, by a continuous approximation $g^{(p)}(t)$, once more shown in fig. 14.32. The result is an approximate energy function

$$E^{(p)} = \sum_{k=1}^{K} (u_{i(k)} - d_k)^2 + g^{(p)}(u_i - u_{i-1}).$$

More specifically,

$$g^{(p)} = \begin{cases} \lambda^2 t^2 & \text{if } |t| < q \\ \alpha - c(|t| - r)^2/2 & \text{if } q \leq |t| < r \\ \alpha & \text{if } |t| \geq r \end{cases}$$

where $c = 1/2p$, $r^2 = \alpha(2/c + 1/\lambda^2)$, and $q = \alpha/(\lambda^2 r)$. The method is then to find the minimum of $E^{(1)}$ which, it can be shown, is a convex approximation to $E = E^{(0)}$. Then from that minimum, the local minimum of $E^{(p)}$ is tracked as p varies from 1 to 0. The claim is that following this progression, optimizing all the way, is a particularly effective strategy. Justification is found both in theoretical analysis, too involved to give here, and practical implementations.

Results for a similar, but not exactly identical regularisation type of approach with discontinuity preserving characteristics are shown for the input image of fig. 14.33. Fig. 14.34 shows the intensity as a 3D profile. Application of the algorithm yields the result shown in fig. 14.35. As can be seen, edges have been sharpened and noise in the different segments has been smoothed. The new 3D profile for the image part of fig. 14.34 is shown in fig. 14.36.

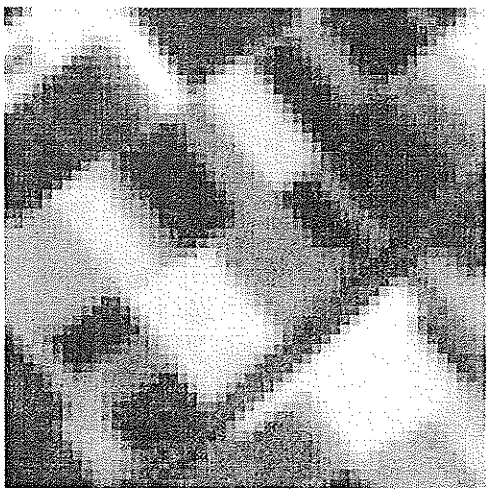

Figure 14.33: *Example image for the regularization approach.*

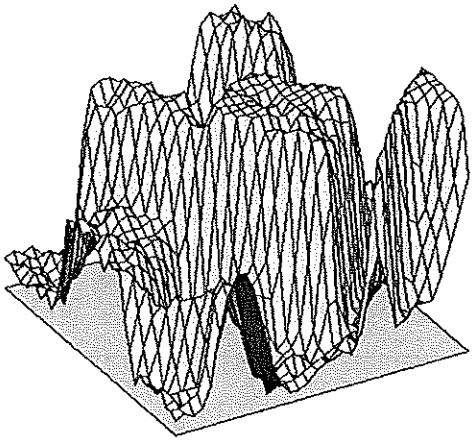

Figure 14.34: *Intensity of portion in fig. 14.33 as 3D plot.*

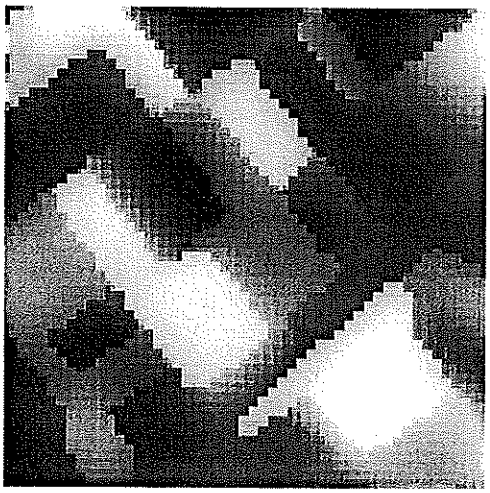

Figure 14.35: *Regularized output for fig. 14.33.*

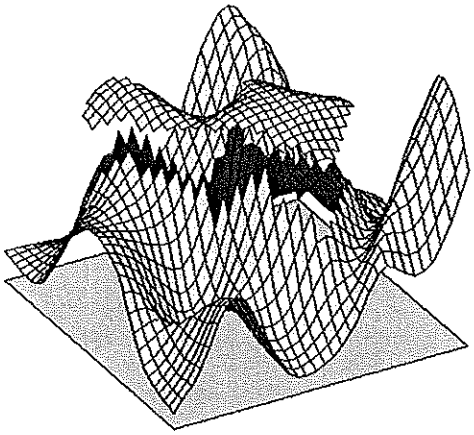

Figure 14.36: *Intensity for the same portion as that of fig. 14.34 as a 3D plot. Notice the clear discontinuities and the smoothing in between.*