An adequate notation should be understood by at least two people, one of whom may be the author. (Abdus Salam)

Chapter 11

Manual Segmentation and Signature Tracking

Numerous applications requiring a high segmentation accuracy exist in various domains. If these applications permit to compute the segmentation masks offline, it can be advantageous to use semi-automatic segmentation techniques. In semi-automatic segmentation, the user controls the segmentation manually, but he is supported by the computer to relieve him from working at the pixel level. A popular approach to semi-automatic segmentation is the Intelligent Scissors tool, which uses shortest-path algorithms to locate the object contour between two user-supplied control points. This chapter proposes a new interactive segmentation algorithm, which is based on the same idea as Intelligent Scissors, albeit providing a user-interface without the need for special control points. Instead of placing control points, the user specifies a rough corridor along the object boundary, in which the computer searches for a shortest circular path. This provides an intuitive user interface, which also supports a natural way to iteratively improve the segmentation by changing the corridor shape. Furthermore, the algorithm is extended with a tracking component, such that an object that has been defined in one image can be automatically segmented in the successive frames. However, the algorithm always allows to interactively intervene in the tracking and provide corrections when the automatic tracking shows errors.

11.1 Introduction

In various applications, the accuracy and reliability of automatic segmentation algorithms is not sufficient. One of these applications is image editing, where objects from one image should be copied into another image. Usually, high-quality results are of primary concern and therefore, it is required to carry out the segmentation manually. Clearly, a manual segmentation is possible even for the most difficult input sequences, but the work is tedious. To relieve the user from working at the pixel-detail level, semi-automatic segmentation algorithms can be used, since they provide a compromise between work-flow efficiency and accuracy of the results. With these algorithms, it is sufficient for the user to coarsely mark the object, while the algorithm extracts the detailed object boundaries at the pixel level.

Various approaches for semi-automatic segmentation have been proposed. They can be coarsely separated in region-oriented algorithms and edge-based algorithms. We have already presented a simple example of a *region-oriented* algorithm in the graph-editor that we described in Chapters 9 and 10. A more advanced region-based algorithm that also supports textured regions, is GrabCut [158]. The principle of all region-based algorithms is that the user places some markers inside the object and in the background. Afterwards, the algorithm examines the color and texture around these markers to separate these regions.

On the other side, there are *edge-based* algorithms, of which the Intelligent Scissors algorithm [130] is the most prominent one. In this algorithm, the user traces along the object as if he would cut it out with a pair of scissors. However, the cutting path is automatically locked to the nearest strong edge in the image, which is most probably the object contour. From time to time, the user places control points to fix the contour found so far. The most frequent problem with this algorithm is that the contour snaps to high-contrast clutter in the background, instead of a lower-contrast object edge. If this is discovered too late, it is difficult to make corrections, since control points have to be moved or inserted.

In this chapter, we propose a new edge-based algorithm, which uses the same concept as Intelligent Scissors, but which does not require the user to place control points. Instead, he draws a coarse *corridor* along the object boundary, and the computer locates the pixel-accurate path around the object within this corridor. The central part of this algorithm is a newly developed shortest circular-path algorithm. This new segmentation algorithm has two advantages: no control points have to be set and the segmentation result can be easily improved if the result is not satisfactory as the corridor can be modified at any time.

Furthermore, we propose an extension to the Corridor Scissors tool for a

more efficient segmentation of video sequences. The algorithms mentioned so far are operating on independent single images, which still requires a large amount of work if an object should be extracted from a video sequence, since every frame has to be processed independently. This manual work can be reduced by *tracking* the user-defined object through the video sequence to relieve the user from redefining the object in every frame. Should there be a tracking error, it can still be corrected by user intervention before the tracking continues with the succeeding frames.

What makes our tracking algorithm special is that it also integrates the texture information of the object that was found in the previous segmentation. Our algorithm extracts the texture information along the border of the object and stores it as the *object signature*. The successive frame is then searched for a deformable contour that shows a similar signature. To find the optimum contour location, we again apply a shortest circular-path algorithm, which now incorporates texture information from the object signature to detect the same object.

In the successive section, we briefly introduce the Intelligent Scissors algorithm and describe typical problems of that approach. This leads us to our proposal of the Corridor Scissors tool. The central algorithm of the Corridor Scissors is a shortest circular-path search, which is described in Section 11.3. Finally, in Section 11.4, we extend the tool with tracking capabilities.

11.2 From Intelligent to Corridor Scissors

11.2.1 Intelligent Scissors algorithm

The Intelligent Scissors tool is an edge-based segmentation algorithm. The user first selects a start point on the object contour, after which the computer continuously computes the minimum-cost path between the start point and the current position. A cost function assigns lower cost to stronger edges in the image, such that the minimum-cost path follows strong contours in the image (Fig. 11.1). If the user is satisfied with the current segment, he places a new control point, which ends the current contour segment and at the same time serves as the new start point. By repeating this process, the user can define the complete object contour step by step.

To compute the minimum-cost path, the Intelligent Scissors algorithm considers the input image as a graph, where each image pixel corresponds to one node in the graph. Graph edges connect nodes that correspond to neighboring pixels. Weights are assigned to these edges according to the inverse gradient strength between the two corresponding pixels. Consequently, strong gradients induce small edge weights. The total path cost

(a) Intelligent Scissors. (b) Edge costs.

Figure 11.1: *In the Intelligent Scissors tool (a), the user first selectes a start point. When he moves the pointer across the image, the computer draws the minimum-cost path between the start point and the current position. The cost function (b) assigns lower costs to stronger edges in the image.*

is defined as the sum of the costs of all graph-edges on the path. After the user has placed the start point for the graph search, the computer can already begin to compute a full tree of shortest paths to all pixels with the Dijkstra algorithm [31]. A single run of the Dijkstra algorithm is sufficient, since it computes inherently all shortest paths from the start position to all other nodes. The attractive aspect is that the path between the current pointer position and the start point can be determined instantaneously by just looking up the minimum-cost path in the full shortest-path tree.

Edge costs

Every edge e_i in the graph representation is attributed with a cost c_i. In the original work [130], this cost was composed of a weighted sum of six different cost components. These include gradient strength, continuity of gradient direction, and object/background color. However, four of them have only a small weight (10% each) and according to our experiments, they have no significant influence on the result. Hence, for simplicity, we only use the two most significant components of the cost function. These two cost components are the gradient strength c_G and a Laplacian zero-crossing detector c_Z. They are defined by

$$c_G = 1 - \frac{||\nabla I||}{\max ||\nabla I||} \quad ; \quad c_Z = \begin{cases} 0 \text{ at Laplacian zero crossings,} \\ 1 \text{ otherwise,} \end{cases} \tag{11.1}$$

where I denotes the greyscale input image. The combined edge costs are defined as $c = (1-\alpha)c_G + \alpha c_Z$, where α is a user-defined weighting factor.

11.2.2 Problems of the Intelligent Scissors tool

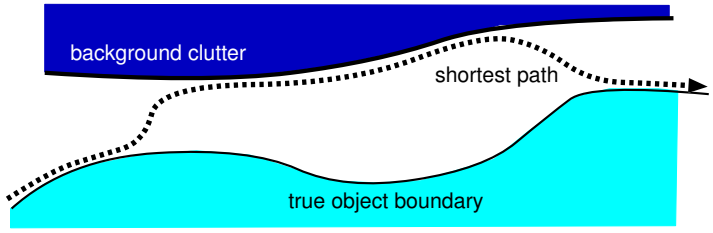
Although the user interface to the Intelligent Scissors algorithm can be understood quickly by most users, there is the inconvenience that the user has to place control-points regularly, because with increasing distance between start and destination point, the extracted path often leaves the object boundary. This effect is especially strong if there are high-contrast edges in the background near the foreground object (Fig. 11.2). In this case, the stronger gradients in the background area lead to a low-cost path. Even though the cost to reach this background clutter may be high, the lower cost in the high-contrast background outweighs a slightly higher cost along the desired object boundary when the path length increases. In the Intelligent Scissors user-interface, this effect shows as a sudden change of the complete path or as an unstable toggling between alternative paths, and the user has to add a control point to stabilize the path again.

As a solution for the problem of background-clutter attraction, it is proposed in [120] to limit the search area to a rectangular area between the seed and destination position. The width of the rectangle is controlled by the user (Fig. 11.2(c)). While this approach may ease the segmentation in some difficult situations, it complicates the interaction process, since another degree of freedom (width of rectangle) has to be controlled by the user.

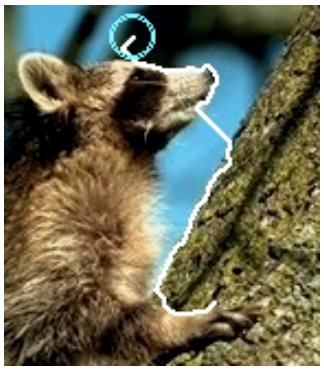
An alternative solution is to use *path cooling*. In this approach, the cost of the edges on the current path are gradually decreased. This means that the parts of the path that remain stable for some time obtain a lower cost, which again decreases the probability that this part of the path is modified later. If a part of the path is stable for a longer time, that part is fixed completely and the start point of the search is moved to the end of this part. This approach eliminates the need to place special control points. However, the speed of user interaction is dictated by the path cooling and the user has to conform to this. Moreover, once the user makes a segmentation error, it is difficult to undo this error, especially if the error is in the already *frozen* part of the path. Finally, this algorithm is also computationally expensive, since a new shortest-path tree has to be computed after each of the frequent cooling steps.

11.2.3 The Corridor Scissors tool

We propose a new segmentation tool, called *Corridor Scissors*, that uses a minimal, yet flexible user interface to define the desired segmentation. The Corridor Scissors algorithm does not require the explicit setting of control points, and it provides a simple and intuitive approach to modify the



(a) The shortest path is distracted from the true object boundary because of near high-contrast background clutter.



(b) Path is distracted by the high-contrast background edge.

(c) Restricting the search area in the *Rubberband* algorithm.

Figure 11.2: *A high-contrast edge in the background that is much stronger than the true object edge may lead to a wrong object contour. Even though the cost to reach this high-contrast edge may be high, this is outweighed by the decreased cost along the contour. The* Rubberband *algorithm (c) proposes to reduce this effect by limiting the area of the graph search to a rectangle between the last control point and the current position.*

Figure 11.3: *Segmentation with low object/background contrast.*

segmentation result until it is satisfactory. Instead of placing control points on the object contour, the user coarsely traces along the object contour with a thick brush. This defines a corridor around the object, in which the true object boundary can be found (Fig. 11.3). After the circular corridor has been defined, the computer searches for a shortest circular path inside of the corridor. The corridor not only prevents that the path is attracted by distant background clutter, but it also reduces computation time since the search space is reduced to the corridor area. If the user wants to improve the segmentation, he can do so by simply changing the shape or width of the corridor. Whenever the corridor shape is modified, a new shortest circular-path search is applied to the corridor area. An example for the problem of snapping to a near high-contrast edge is shown in Figure 11.4. First, a very wide corridor also covers a high-contrast edge that is not the object edge. Interrupting that path by narrowing the corridor forces that a different path (the correct object contour) is taken.

The underlying algorithm of Corridor Scissors is based on the same graph-search approach as the Intelligent Scissors algorithm. However, instead of searching for ordinary shortest paths, an algorithm for computing the shortest *circular* paths has to be applied. A new algorithm for the shortest circular-path problem, which has a comparable computation time to an ordinary shortest-path search, is presented in Section 11.3.

11.2.4 Experiments and results with Corridor Scissors

Some segmentation results that were obtained with the Corridor Scissors algorithm are depicted in Figure 11.5. The left column shows the input images and the right column the extracted objects, respectively. Superim-

(a) Path is distracted to strong gradient.

(b) Interruption of the wrong path leads to the correct segmentation.

Figure 11.4: *Improving a segmentation with Corridor Scissors. If the path is distracted to a higher contrast edge (a), the corridor can be modified to make this path impossible (b).*

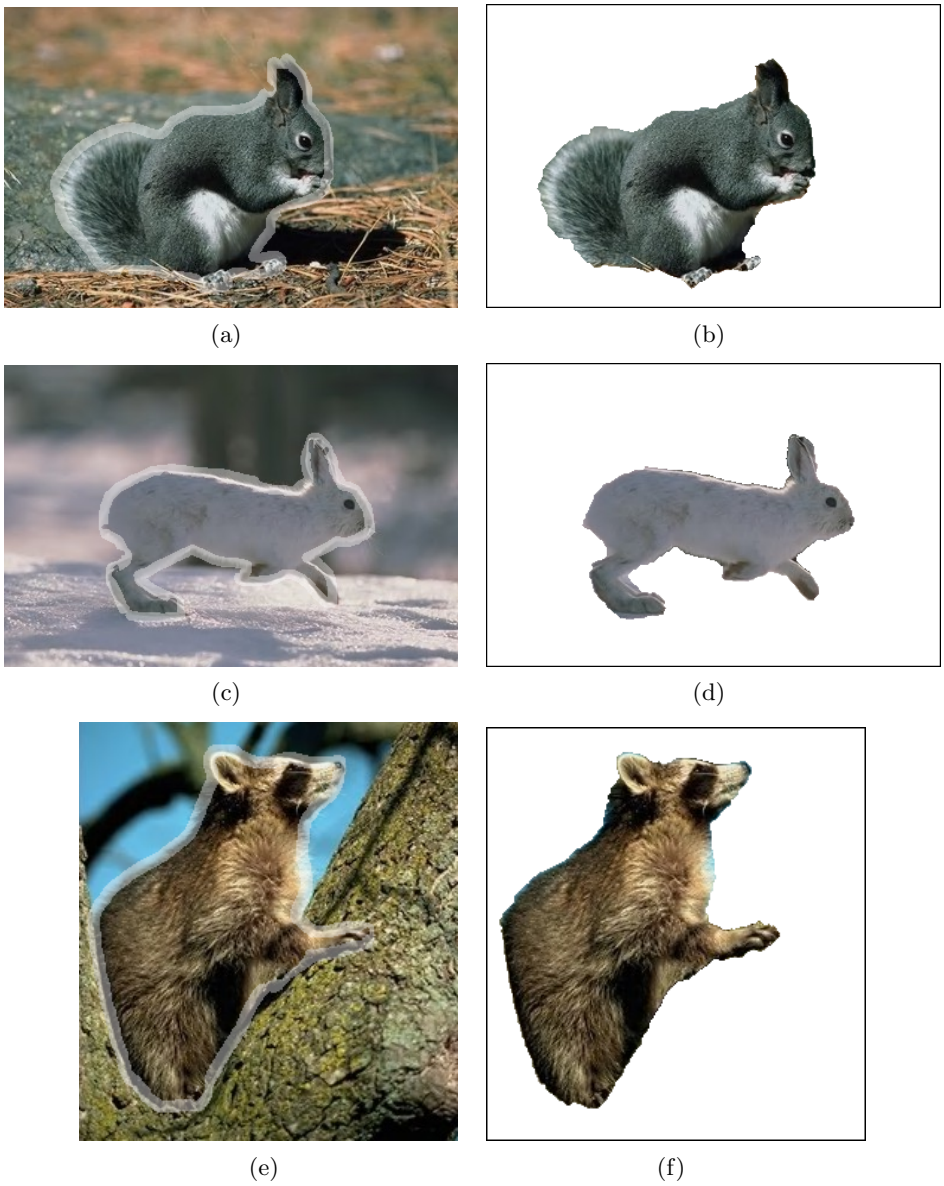
posed onto the input images are the corridors that were used to obtain the results on the right-hand side. Note that in the image of the squirrel (a), there is fine texture at the ground and the contour along the hairy tail is difficult to define. In the image of the rabbit (c), the color of the foreground object is close to the background color.

It can be seen that the segmentation results were obtained with almost no manual correction of the corridor. Only at the ears of the squirrel and the rabbit, as well as the tail of the rabbit, the corridor was made a bit smaller to get the correct contour.

Our experience with a large variety of input images is that the objects are generally easy to define. The only difficult case are objects with long thin structures like antennas, since these thin objects are usually covered completely by the corridor. In this case, the algorithm prefers to simply let the path cross the thin object instead of following the long contour on both sides of the object. These errors are difficult to correct because the only possibility is to trace both sides of the object contour with the corridor, without touching itself (no bridge between both sides should be built).

11.3 Shortest circular paths

The Corridor Scissors algorithm and also the tracking algorithm which is described in Section 11.4 are built upon an algorithm to compute shortest circular paths in graphs. This section proposes a new algorithm with low computational complexity.



(a) (b) (c) (d) (e) (f)

Figure 11.5: *Segmentation results for the Corridor Scissors algorithm. The marked corridor is depicted on the left, the segmentation result is on the right. Note the changes of the original corridor to improve the segmentation result (e.g., at the ears of the squirrel).*

11.3.1 Definition of circular paths

In the Corridor Scissors application, it seems intuitive to understand what we mean by a *circular path*. However, in the following, we develop an algorithm that works on general planar graphs. Hence, a clear definition of circular paths is required. For this reason, we define circular paths formally using the following two definitions.

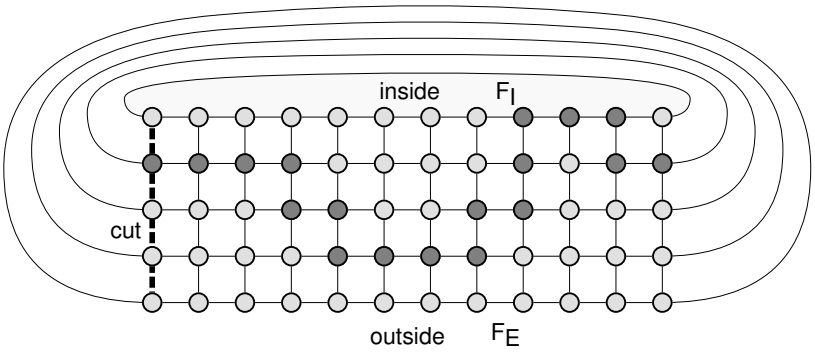
Definition (cut path): Let $G = (V, E)$ be a planar graph with nodes V and edges E. Furthermore, the graph is assumed to be embedded in the plane with two of its faces denoted as F_I, F_E, corresponding to the *inner hole* of the graph, and the *exterior area* (see Fig. 11.6). We call any path connecting F_I and F_E a *cut path* $\bar{p}$.

Definition (circular path): Let G again be a planar graph with two faces labeled F_I, F_E. We define a circular path on the graph G as a cyclic path $\dot{p} = v_i \rightsquigarrow v_i$, such that $\dot{p}$ and any cut path $\bar{p}$ have at least one node in common.

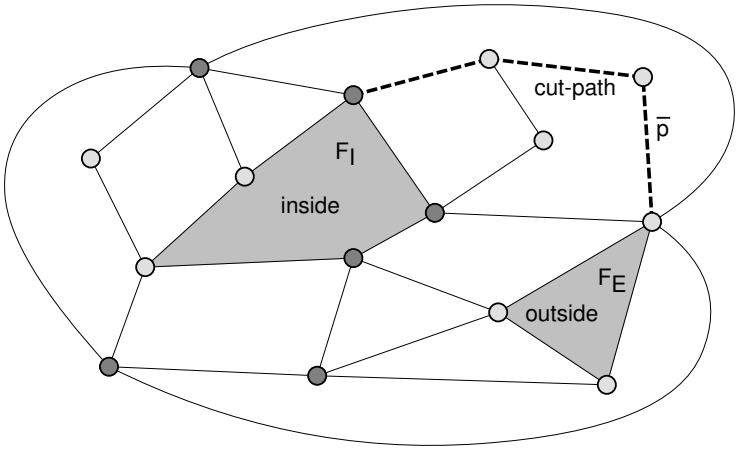
Two examples of graphs are depicted in Fig. 11.6. The first is a regular grid graph similar to those types of graphs that occur in the Corridor Scissors algorithm. These graphs have two large faces which are the inside area (the object to be segmented) and the exterior area (the background). The second example is a general planar graph, in which the inside and exterior faces have been specified in order to be able to define circular paths. These general graphs usually do not occur in our application, but we provide the example because our algorithm also works on these general planar graphs.

Instead of computing a shortest circular path on the original graph, it is convenient to transform it first into a *corridor graph*. The construction of the corridor graph can be imagined as cutting the original ring-shaped graph apart along a cut path apart to get a lane-shaped graph. Formally, this construction can be described as follows.

Construction (corridor graph): Let $G = (V, E)$ be a planar graph and $\bar{p} = v_1 v_2 \cdots v_N$ a cut path between F_I and F_E, consisting of the nodes $\bar{V} = \{v_1, \ldots, v_N\}$. Furthermore, we assume the following property of G: if V_n is the set of nodes adjacent to nodes in $\bar{V}$, then V_n comprises exactly two connected components V_l, V_r. If G has this property, we can transform G into a *corridor graph* $G' = (V', E')$ as follows. We set $V' = V \cup \bar{V}'$, where $\bar{V}' = \{v'_1, \ldots, v'_N\}$ is a copy of $\bar{V}$. The edges are defined as

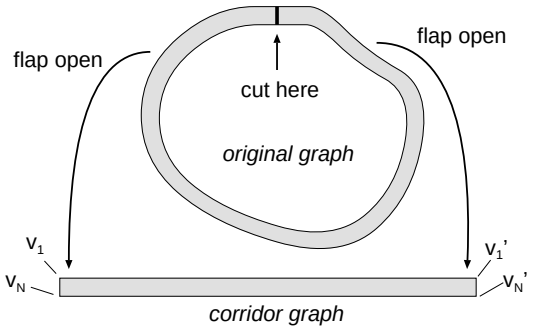


(a) Circular-grid graph.

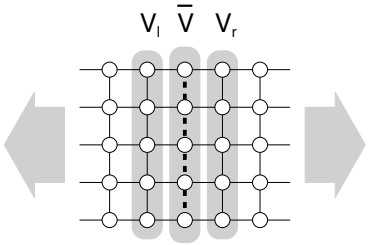


(b) General planar graph.

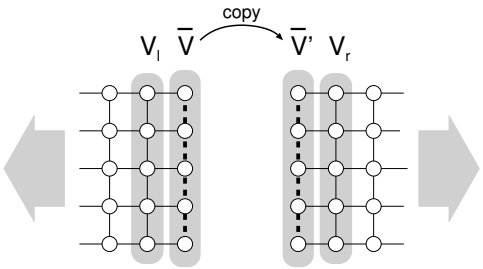
Figure 11.6: *Two graphs with examples of valid circular paths (dark nodes). In each, an example cut path (path between inside and outside face) is marked. A circular path is defined as a cyclic path that crosses any arbitrary cut path.*



(a) The circular input graph is cut into a lane-shaped corridor graph.



(b) The nodes on the cut and the left and right neighbors.



(c) The nodes on the cut are duplicated and the graph is cut apart.

Figure 11.7: *The input graph is cut apart along a cut path to get a corridor graph.*

$$E' = (E \underbrace{\setminus (V_r \times \bar{V})}_{\text{cut graph between } V_r \text{ and } \bar{V}}) \cup \underbrace{\{\{v_i \in V_r, v'_k \in \bar{V}'\}}_{\text{reconnect } V_r \text{ with } \bar{V}'} \mid \{v_i, v_k\} \in E\}. \qquad (11.2)$$

This copies the nodes $\bar{V}$ on the cut path to $\bar{V}'$ and reconnects the adjacent nodes V_l and V_r such that V_l connects to only $\bar{V}$ and V_r only to $\bar{V}'$. The construction is visualized in Fig. 11.7.

Most currently known algorithms for shortest circular paths first transform the input graph to a corridor graph. On the corridor graph, searching for a circular path can be described easily as searching for a path from the "left side" $\bar{V}$ to the right side $\bar{V}'$ with the additional constraint that a path starting at v_i must end in v'_i. Even though a general planar graph can be transformed into a corridor graph by cutting along any arbitrary cut path, care must be taken because shortest circular-path algorithms that operate on the corridor graph can only find paths that do not cross the cut path more than once. One way to minimize this risk is to choose a cut path which is as short as possible. In Section 11.3.2, we will present a safe technique to find a cut path that is guaranteed to be crossed only once.

11.3.2 Computation of shortest circular paths

Previous work

In [178], several algorithms for the circular-path search are proposed. The *Multiple Search Algorithm* (MSA) uses $|\bar{V}|$ independent runs of the Dijkstra algorithm, where $|\bar{V}|$ is the length of the cut path. Each run uses fixed, opposing seed and destination nodes at both ends of the graph. After these runs, the result that gives the minimum cost is selected. This algorithm always finds the optimal solution, but it is computationally intensive because the Dijkstra algorithm has to be executed $|\bar{V}|$ times independently over the full graph.

The *Image Patching Algorithm* [178] only gives an approximate solution, but requires less computation time. In this case, the corridor graph is enlarged by appending part of the graph from each end to the opposite side. An ordinary shortest path is then computed through the complete, patched graph. The part of the shortest path that lies inside of the original graph is extracted and assumed to be the optimal circular path. However, even though this heuristic works in many cases, it is not assured that the optimal circular path is found. Moreover, the algorithm can even lead to non-cyclic paths. The quality of the result can be increased by enlarging the patched areas, but the required patch size is not known and the computation time increases. It is also easily possible to construct examples in which the found path remains non-circular for arbitrarily long patches.

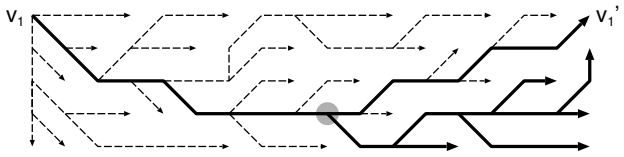


Figure 11.8: *Example tree of shortest paths as obtained with the Dijkstra algorithm. Consequently, there is a node (marked in the image) at which all paths to the nodes on the right side are rooted.*

Finally, a branch-and-bound algorithm has been proposed in [7] that, on the average, requires only $\log_2 |\bar{V}|$ runs of the Dijkstra algorithm on the input graph. But the worst-case still requires $|\bar{V}|$ runs of the Dijkstra algorithm over the full graph.

For planar graphs, a different approach is to view the shortest circular-path problem as a maximum-flow problem. By adding two dummy nodes inside each of the two faces F_I and F_E, searching for the shortest circular path is equivalent to a maximum-flow problem between the two dummy nodes. However, maximum-flow algorithms have a high computational complexity of $O(|V|^3)$ for standard preflow-push algorithms, or $O(|V| \cdot |E| \log(|V|^2/|E|)) = O(|V|^2 \log |V|)$ for the fastest algorithm currently known [77]. It should also be considered that this fast algorithm requires a complex implementation.

Preliminary considerations

Our algorithm for computing the shortest circular paths is built upon the Dijkstra algorithm for ordinary shortest paths. In order to show the correctness of our algorithm, we will subsequently exploit the following special property of the Dijkstra algorithm.

Property: The Dijkstra algorithm always builds a complete shortest path tree, rooted at the seed node (Fig. 11.8). Hence, a single run of the Dijkstra algorithm does not give only the shortest path to the specified destination, but also the shortest path to every other node.

Additionally, we need the following theorem about shortest paths that is easy to derive.

Theorem: Let $G = (V, E)$ with $V = \{v_i\}_i$ be a (not necessarily planar) graph and let $u = v_{i_1} v_{i_2} \dots v_{i_n}$ and $w = v_{k_1} v_{k_2} \dots v_{k_m}$ be two minimum-cost paths (see Fig. 11.9). Then we can state that if u and w have two nodes v_p and v_q in common, there is a path $w' = v_{k_1} \dots v_{k_m}$ with the same cost as w and which has a common subpath $s_u = v_p \rightsquigarrow v_q$ with path u.

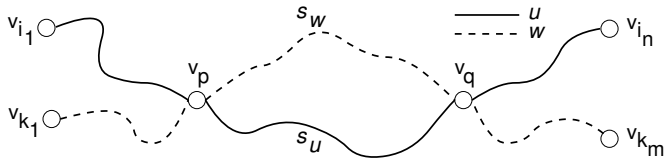


Figure 11.9: *Two minimum-cost paths with at least two common nodes share the whole subpath between the common nodes.*

Proof: If s_u and s_w have equal cost, there is nothing to show. Hence, assume that the two subpaths s_u, s_w between v_p and v_q have different cost. Then, the cost of either s_u or s_w must be lower than the other. Let us assume that the cost of s_u is lower than the cost of s_w. Consequently, w cannot have minimum cost, because the cost can be lowered by replacing the subpath s_w with s_u. □

As a direct consequence of this theorem, we can state the following two corollaries, which will be used in the development of the shortest circular-path algorithm.

Corollary 1: Minimum-cost paths may cross at most once.[1]

Corollary 2: If two paths share a common seed, then both paths will share a common subpath to their destination until both paths split. After the split, the paths will not cross.

In other words, paths with disjoint endpoints may cross once, while paths with one endpoint in common will never cross.

New shortest circular-path algorithm

We now define a fast algorithm for computing shortest circular paths by exploiting the properties of the Dijkstra algorithm and the above-mentioned theorem. The complete circular-path algorithm is divided into two parts. In almost all practical cases, the first part of the algorithm can already compute the optimal circular path and the second part of the algorithm can be omitted. In the practically very rare case that the first part cannot compute the optimum path, this is detected by the algorithm and the second part of the algorithm is used instead. This second, alternative algorithm is more computationally complex, but provides an optimal solution in all cases.

[1] In fact, minimum-cost paths can cross more than once, but there is always a path of similar cost that does not cross more than once. Hence, when searching for a minimum-cost path, we can assume that they do not cross more than once.

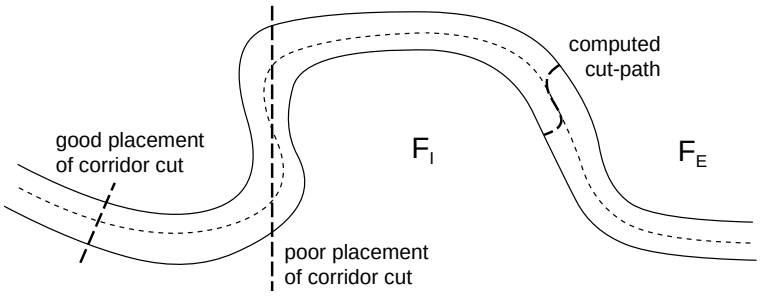


Figure 11.10: *Corridor cuts should be placed at narrow positions, perpendicular to the corridor. Note that the shortest path cannot cross the cut more than once. We determine a cut path by computing a shortest path between the two faces F_I, F_E. This ensures that it is crossed only once.*

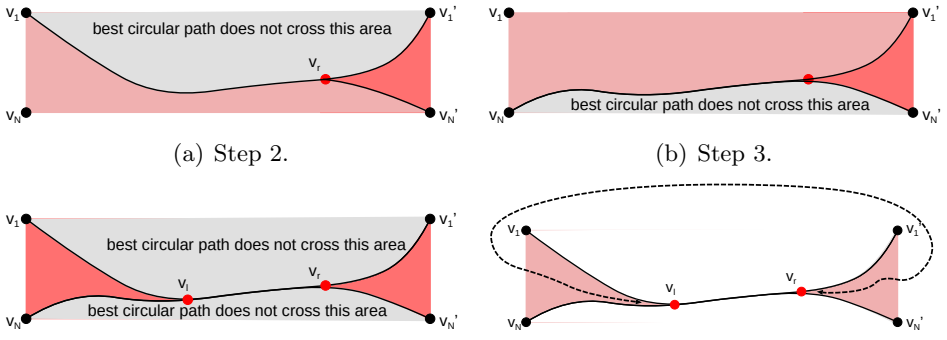
The algorithm is based on the observation that the corridor graphs in our application are usually much longer than they are wide. Furthermore, the data within the graph usually shows relatively clear low costs on a path along the corridor, since the corridor is placed on the object border. Consequently, the shortest circular-path problem is not difficult in the sense that the approximate path is clear. Usually, at some distance from the cut, there is only one "main route" to follow, only close to the cut itself, it is difficult to predict the path position (see Fig. 11.26). Hence, our algorithm tries to identify a common subpath along the corridor that all circular shortest-paths share. Once this subpath is known, only the ends of this subpath have to be connected to form a circular path.

Algorithm part I

The first part of the algorithm (in the following called AP1) assumes that a common subpath can be found in the corridor. If this is not the case, this will be detected and part 2 of the algorithm (AP2) will be used. AP1 comprises the following four computation steps.

AP1-Step 1: Transform the input graph into a corridor graph. In determining the cut path, it should be noted that the cut path can only be crossed once by the shortest circular path. Hence, a cut in an acute angle along the corridor should be prevented (see Fig. 11.10). One simple heuristic solution to this problem is to place the cut such that the cut path is as short as possible (minimum number of nodes in $|\bar{V}|$).

However, it is possible to determine a cut path that will not be crossed by the shortest circular path more than once as follows. Compute a minimum-



(a) Step 2. (b) Step 3.

(c) Situation after Step 3. (d) Step 4.

Figure 11.11: *Illustration of Steps 2-4 of AP1. (a) A shortest path tree is computed from position v_1. This gives all shortest paths to all nodes in $\bar{V}'$. All these paths share a common subpath up to a node v_r. (b) A shortest-path tree is computed from position v_N. Similarly as in (a), all shortest paths to $\bar{V}'$ share a common subpath. (c) Both trees computed in Step 2 and 3 join at a common node v_l. Because the shortest circular path cannot pass the area above $v_1 \rightsquigarrow v_1'$ and below $v_N \rightsquigarrow v_N'$, it must contain the subpath $v_l \rightsquigarrow v_r$. (d) Since a subpath of the shortest circular path is known, only the connection from v_r to v_l through the grey area has to be computed.*

cost path $\bar{p}$ from the nodes of the face F_I to the nodes of the face F_E. By definition, this is a cut path. If the shortest circular path crosses the cut path $\bar{p}$ more than once, then this is a contradiction to the above theorem, since both paths have minimum cost. Consequently, the computed cut path is only crossed once. Note that it is sufficient to start the computation of the minimum-cost path at only one arbitrary node of F_I instead of considering all nodes of F_I as start node. This significantly reduces the computation time required to find a cut path.

AP1-Step 2: Perform a Dijkstra-search beginning at node $v_1 \in \bar{V}$, which is the top-left node of the corridor. This pass will compute at the same time shortest paths to nodes v_1' and v_N' (see Fig. 11.11(a)). Since the starting point is shared, both paths will share a subpath up to a node v_r. Shortest paths from the left side of the corridor to the right side cannot traverse the area above $v_1 \rightsquigarrow v_1'$, since this would mean that they have to cross $v_1 \rightsquigarrow v_1'$ twice, which is not allowed because of the above theorem. Hence, all nodes above the path $v_1 \rightsquigarrow v_1'$ can be ignored in the following steps.

AP1-Step 3: Perform a second Dijkstra-search from node v_N to node v_N'.

In almost all practical cases of our application, especially if the corridor is long compared to the corridor width, this path will join the shortest paths $v_1 \rightsquigarrow v_1'$ from the last step at some node v_l, where v_l is closer to the start as v_r (see Fig. 11.11(b)). If this is the case, then we are sure that all shortest paths between the left side and the right side share at least the subpath $v_l \rightsquigarrow v_r$. In the other case that both paths do not join, we cannot use the simple AP1 and have to switch to the more general AP2 that is described below.

AP1-Step 4: Since it is already known that the subpath $v_l \rightsquigarrow v_r$ is part of the shortest circular path, we only have to search for a connection from v_r back to v_l to close the cycle. We also know that this connection must lie between the area of the previously computed shortest paths. Hence, we perform a third Dijkstra-search from v_r to v_l over the nodes in the shaded area depicted in Fig. 11.11(d). Appending this path to the path $v_l \rightsquigarrow v_r$ gives the shortest circular path $v_l \rightsquigarrow v_r \rightsquigarrow v_l$.

Note that the search for $v_r \rightsquigarrow v_l$ can be implemented more efficiently by searching backwards from $v_l \rightsquigarrow v_r$. In this case, the search can be restricted to the left area, since the shortest-path tree for the right area, rooted at v_r, is already known from the previous search of Step 3.

Algorithm part II

The second part of the algorithm (AP2) is a more complex, generalized version of AP1. It is used if AP1 detects that there is no common subpath for all paths along the corridor. In this case, we have the situation of Fig. 11.12. The shortest path $v_1 \rightsquigarrow v_1'$ (a) and the shortest path $v_N \rightsquigarrow v_N'$ (b) is known from the computations of AP1. Both paths are disjoint, because otherwise AP1 would have found the solution.

AP2 uses a recursive approach to continuously split the graph into smaller graphs until the shortest circular-path problem can be solved similarly as in AP1.

The input of each recursion is an input graph which is bounded to the top and bottom by shortest paths (a) and (b). Let us denote the top bounding path (a) as $v_a \rightsquigarrow v_a'$ and the bottom bounding path (b) as $v_b \rightsquigarrow v_b'$. The algorithm is initialized with $v_a = v_1$ and $v_b = v_N$, as shown in the figures. Note that both paths are circular paths. We now compute a shortest-path tree, starting from the middle node $v_c = v_{(a+b)/2}$ at the left side of the graph (Fig. 11.13). In one run of the Dijkstra algorithm, we obtain all shortest paths to the nodes between v_a' and v_b' at the right side. Note that we can limit the computation to the area between the two bounding paths (a),(b).

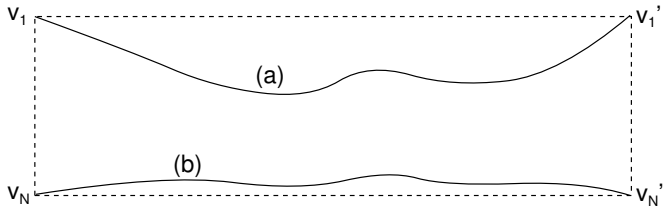


Figure 11.12: *Situation at beginning of AP2. The disjoint shortest paths (a) and (b) are known.*

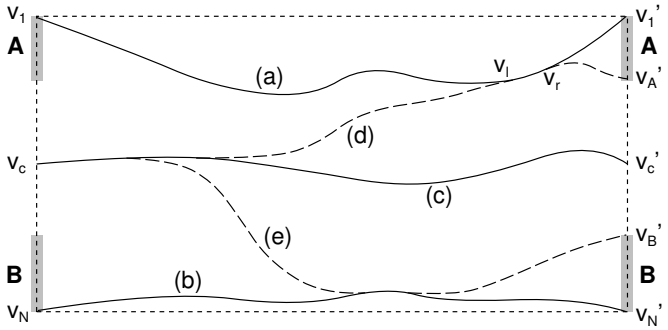


Figure 11.13: *A shortest-path tree is computed, starting at v_c, which is the node in the middle of the left side. The bottom-most node v'_i on the right side, for which $v_c \rightsquigarrow v'_i$ joins the path (a) is denoted as v'_A and the corresponding path as (d). Similarly, we obtain v'_B as the top-most node on the right side, for which $v_c \rightsquigarrow v'_B$ (e) still joins the path (b). Furthermore, we have computed the circular path $v_c \rightsquigarrow v'_c$ for the center node v_c.*

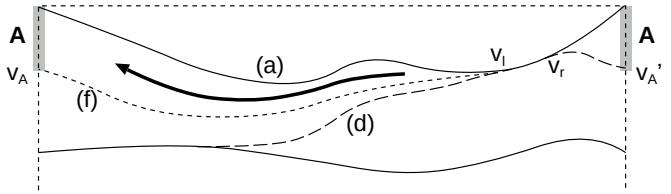


Figure 11.14: *Given (a) and (d), we know that all shortest circular paths in range* **A** *include the subpath $v_l \rightsquigarrow v_r$. Hence, we search for a shortest path starting from v_l in the indicated direction to v_r. This is the shortest circular path within range* **A**. *Note that this search can be restricted to the area between (a) and (d). The path (f) is the shortest circular path $v_A \rightsquigarrow v'_A$, which will be used as upper-bound path to further reduce the graph size in the recursion step.*

If we consider the shortest paths between v_c and the right side of the graph, we see that the path $v_c \rightsquigarrow v'_a$ obviously joins with (a). As we consider destination nodes v'_k further down ($k > a$), there is generally a point from which onwards the shortest path does not join (a) anymore. In fact, there is a node v'_A, $A < c$ which is the last node (from top to bottom) for which the shortest path $v_c \rightsquigarrow v'_A$ touches (a). Let us denote this path as (d), like it is depicted in Fig. 11.13. Note that This is a similar situation as in AP1. Any circular path $v_i \rightsquigarrow v'_i$ with $a \leq i \leq A$ is known to include the common subpath of (a) and (d). This range is identified as $\mathbf{A} = [a; A]$. We can now compute the shortest circular path that starts and ends within range **A** in a single step. To close the shortest circular path within the range **A**, we compute the connection between v_l and v_r similarly to AP1-Step 4 (see Fig. 11.14).

A symmetrical process can be carried out to find the shortest circular path for the area **B** at the bottom of the considered graph. When the shortest circular path in each of the ranges **A** and **B** are known, we have to further consider only the remaining range in between (Fig. 11.15). Since we also know the path $v_c \rightsquigarrow v'_c$, denoted as (c), we can split the problem for the remaining range recursively, by first considering the graph between (a) and (f) with start nodes $v_{A+1}, \ldots, v_{c-1}$, and similarly the graph between (c) and (g) with start nodes $v_{c+1}, \ldots, v_{B-1}$. Once all shortest circular paths for all ranges of nodes along the cut are known, we simply select the shortest of them as the global solution.

Step-by-step examples of the algorithm are provided in the appendix at the end of this chapter.

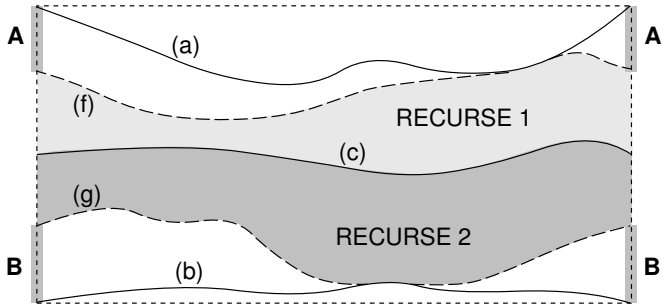
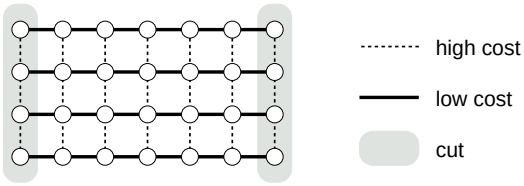


Figure 11.15: *Regions* **A** *and* **B** *are already processed, only shortest circular paths in the range in between are unknown. This range is processed recursively, first processing the graph between (f) and (c), and then the graph between (c) and (g).*



(a) The worst-case.

(b) A nearly worst-case input.

Figure 11.16: *In the worst-case, all* $|\bar{V}|$ *shortest circular paths starting in the cut are disjoint (a). For testing, we used also nearly worst-case inputs in which some noise is added to the worst-case pattern (b).*

11.3.3 Computational complexity

Let us now examine the computational complexity of our shortest circular-path algorithm. In the case that AP1 succeeds, three ordinary shortest-path searches have to be conducted. Using the Dijkstra algorithm with a heap implementation, each search takes $O(|V| \log |V|)$ time, since the graph is planar. Hence, the total running time for AP1 is also just $O(|V| \log |V|)$. Note that in almost all practical cases in our application, AP1 is already sufficient.

The computation time of AP2 is more complicated to determine, because it is directly depending on the input data. In the worst-case (see

Fig. 11.16(a)), every shortest circular path crossing the cut is independent and must be computed. However, because the size of the graph in the recursion is only about half the size of the input graph, computation of shortest paths becomes faster in later stages of the recursion. This gives a total number of nodes to be processed of approximately

$$\sum_{i=0}^{\log|\bar{V}|} 2^i \frac{|V|\log|V|}{2^i} = |V| \cdot \log|V| \cdot (\log|\bar{V}| + 1). \tag{11.3}$$

Furthermore, nodes from the ranges **A** and **B** can be excluded, so that the actual computation time is usually lower. Note that the cut is determined such that it is short, which makes $|\bar{V}|$ small. For increasing image resolution, the corridor area $|V|$ increases quadratically compared to the length $|\bar{V}|$, which gives us a worst-case computation time of $O(|V|\log^2|V|)$.

In practice, the number of recursion steps that have to be carried out depends on the complexity of the data in the corridor. If there are clearly superiour paths within the corridor instead of equally good disjoint paths, a large number of possible paths can be excluded in each step. Since the global complexity of the content in the corridor does not increase with increasing image resolution, we can assume the number of recursions constant. Furthermore, the longer the corridor is compared to its width, the higher the probability that paths join. This leads to a total time for practical data of only $O(|V|\log|V|)$.

Grid graphs

One special, but important case are regular graphs with directed edges, as shown in Figure 11.18. For this type of graph, the shortest-path search using the Dijkstra algorithm can be replaced by a more simple dynamic programming approach, which processes the nodes column by column. This reduces the computation time for an ordinary shortest-path search to $O(|V|)$ instead of $O(|V|\log|V|)$ for the Dijkstra algorithm on general planar graphs. In total, we obtain a practical computation time of also only $O(|V|)$ for the shortest circular-path search and a worst case of $O(|V|\log|V|)$.

Experimental verification

To justify the estimation of computation time, we conducted experiments with different types of inputs. In the first experiment, we took the picture shown in Fig. 11.27 as input graph with costs taken from the pixel luminance. The shortest circular path horizontally crossing the image was computed for different sizes of the input image. Note that this input is

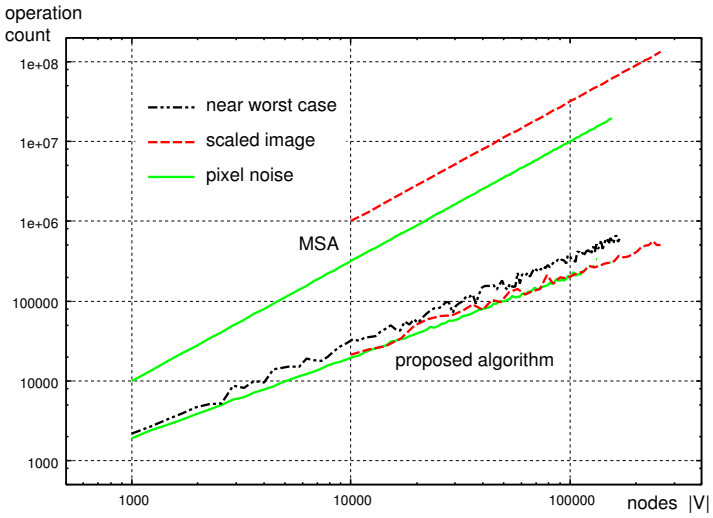


Figure 11.17: *Computation times for input graphs on different types of complicated data. For these complex examples, the computation time of MSA is $O(|V|^{3/2})$, whereas it is only $O(|V|)$ for the proposed algorithm.*

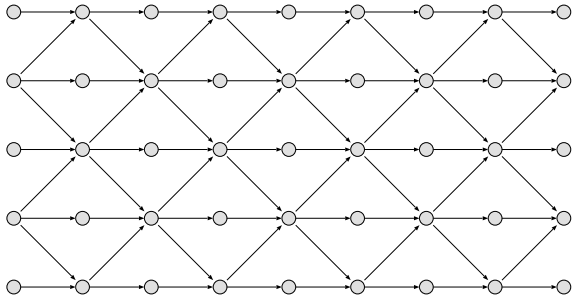


Figure 11.18: *A planar graph on a regular grid structure. With directed edges, shortest paths can be found in time $O(|V|)$ using a dynamic programming approach, compared to $O(|V| \log |V|)$ when applying the Dijkstra algorithm.*

more complex than the usual case in the Corridor Scissors algorithm, since the input image is square and thus, the probability that paths across the image share a common subpath is lower than for the long and narrow corridors. Moreover, the image content is more complex than in the Corridor Scissors application, because in the latter case, the input covers mainly a single object contour.

In the second experiment, we used rectangular grid graphs that are 10 times longer than wide. The edge weights were set to random values. This is also more complex than the case for Corridor Scissors, since the input data shows no structure that increases the probability of common subpaths.

In a final experiment, we synthesized an input that is close to the worst-case. We generated input images with alternating high-cost and low-cost rows with an average cost difference of σ and we added uniform random noise with an amplitude of $\sigma/2$. Note that the actual choice of $\sigma > 0$ has no influence on the shortest paths found.

In all experiments, we used edges arranged as in Fig. 11.18, such that a fast shortest-path search with dynamic programming can be applied. We measured the number of operations, which we defined as the number of nodes processed, for the proposed algorithm and the MSA algorithm. The measured computation times for both algorithms are depicted in Figure 11.17. It is clearly visible that the MSA algorithm has a complexity of $O(|V|^{3/2})$, while the proposed algorithm runs in only $O(|V|)$ time on normal inputs. This is remarkable, since the considered input is more complex than the usual input in practice. A computation time that comes close to the worst case could only be reached with the synthetically constructed input. Hence, we can conclude that even though the worst-case performance of the proposed algorithm is $O(|V| \log |V|)$, the computation time is linear in all practical cases.

11.4 Signature tracking

The Corridor Scissors algorithm described in Section 11.2 is a still-image segmentation algorithm. To process a video sequence, every frame has to be segmented separately. In this section, we propose an extension to our Corridor Scissors algorithm that provides object-tracking capabilities. With object tracking, it is no longer required to edit each frame independently. Instead, a contour that has been defined once can be tracked through the sequence automatically. However, in case of tracking errors, our algorithm supports the manual intervention to correct the segmentation result.

11.4.1 A first tracking algorithm

A popular approach to implement contour tracking is to search the local neighborhood of the previous contour for the new contour. Depending on the internal representation of the contour, different tracking algorithms have been proposed. See, for example, [11] for active contours, [138] for tracking with a level-set representation, or [197] for graph-cut based active contours. The algorithm proposed in [197] can essentially also be applied to our Corridor Scissors framework. By replacing the proposed graph-cut contour detection with our shortest circular paths, we obtain the following simple tracking algorithm.

Let us assume that the motion of the object is so small that the largest distance between contour points in both images is less than w pixels. If we copy the contour found in the previous frame $t-1$ to the current frame t, and if we augment this contour to a corridor to a width of $2w$ pixels, then this corridor contains the object contour in the new frame. To find the new object contour, we only have to apply the previously described Corridor Scissors algorithm in this new corridor. The resulting contour can again be used to generate a new corridor for the successive frame $t+1$, and so on. Note that the user can always intervene this process whenever a tracking error occurs, by simply adapting the shape of the corridor.

11.4.2 Signature tracking algorithm

The above tracking algorithm is very simple to implement and easy to use, but its tracking robustness is low, since the contour in the new image is often distracted by background clutter. In this case, the contour quickly drifts away from the correct object contour and it cannot be found back, as no information about the object itself is used. However, note that the tracking algorithm does not use all the information available from the previous frame. In fact, we can also use the colors along the object contour as an additional source of information about the object's appearance. By searching for exactly the same colors along the object contour as in the original object, the object-contour detection is more robust, since it can distinguish background texture from foreground. We denote this texture information along the object contour as the object *signature*.

Since pixels exactly at the border between foreground and background do not provide stable information, we extract for each pixel along the contour a small block of the texture surrounding this pixel. This results in a signature vector with the length of the object contour, where each entry is a texture block from that contour position. The concept is now to use this signature for replacing the matching cost in the shortest circular-

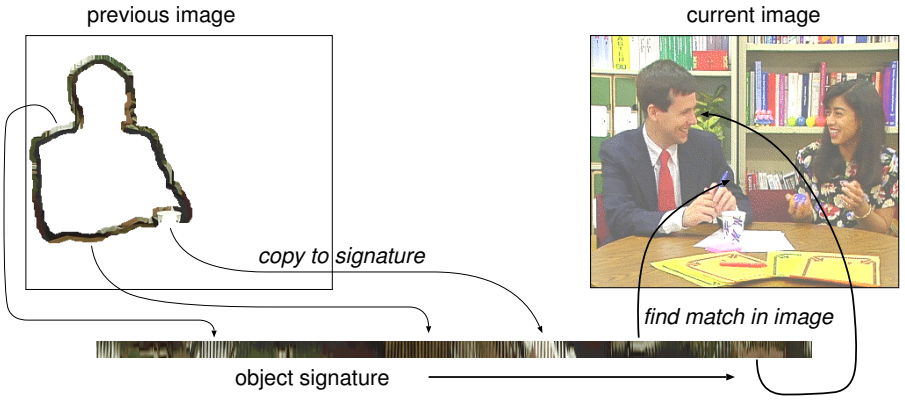
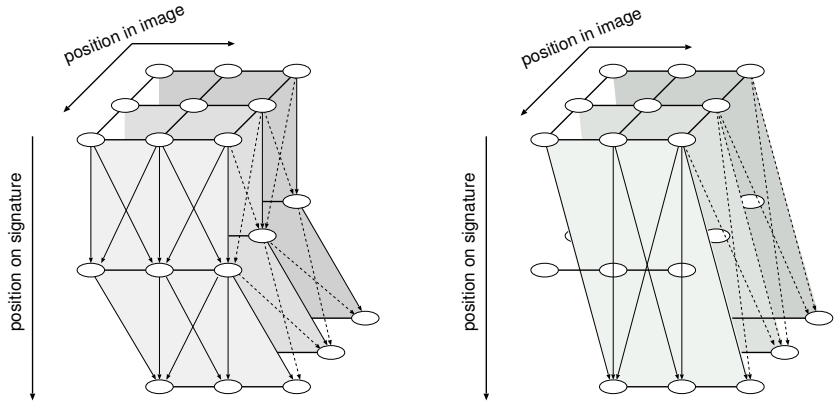


Figure 11.19: *The texture around the object is stored in an* object-signature *vector. In the successive frame, we search for a closed contour showing a similar signature along the object border.*

path search and to find a new object contour which has a similar signature (Fig. 11.19). The additional information from the signature helps to prevent that the circular path snaps to high-contrast background clutter, since the image content along the contour should fit to the recorded signature. One problem in the matching is that the object contour in the new image may be larger or smaller, because of object deformations. Hence, we have to allow for some stretching or shrinking of the signature vector. This change of length is generally non-uniform, i.e., part of the object may become larger, while another part may get smaller at the same time.

The shortest circular-path search now operates on a graph that we can visualize best in three dimensions. Two dimensions (x, y) correspond to the spatial pixel position, just as in the Corridor Scissors above. However, we introduce a third dimension (z), which corresponds to the pixel position on the signature. In this sense, we denote the graph nodes as $V = \{v_{x,y,z}\}$. Each graph node $v_{\{x,y,z\}}$ is attributed with the matching cost between the texture block at (x, y) in the current frame, and the block that was saved at signature position z. If the size of the object contour would not be allowed to stretch or shrink, every step in the x/y plane should be reflected by a step in z direction. But since the object contour in the new image may be larger, we also allow to make a step in the x/y plane without advancing in the z direction. This corresponds to staying at the same contour position in the original image, while advancing in the current image. On the other hand, to be able to shrink the object contour, it is allowed to advance two steps in the z direction in only one step. For all of these possibilities, edges are introduced in the three-dimensional graph (see Fig. 11.20). Because



(a) Edges between successive layers.

(b) Edges skipping one layer.

Figure 11.20: *A small part of the signature search-graph (Fig. 11.21). Directed edges from one layer to the next (a) are the standard links. Taking this edge corresponds to going to a neighboring pixel in the image and making a step to the next position on the signature. Additionally, edges* within *one layer represent steps in the image, without proceeding on the signature (the contour is stretched). Finally, edges skipping one layer (b) make a step in the image and skip one position in the signature (the contour is shrunk).*

the signature is taken from a closed contour, edges are also added from the last layer of z to the first.

11.4.3 Circular-path search with object signatures

Searching for the new object contour can now be carried out again with a shortest circular-path search but now within the presented 3-D graph. Note that in this graph, the edges in z direction are directed, since it is not allowed to move backwards on the signature. The search in this 3-D graph poses two main difficulties. First, the number of nodes in the full three-dimensional graph is so high that the computational complexity becomes impractical. Second, the graph is non-planar, and the shortest circular-path algorithm above cannot be applied. An MSA algorithm would still be possible, but it would increase the computation time even further. We approach these two problems by first significantly reducing the size of the graph, and subsequently using a very fast approximation of the above shortest circular-path algorithm.

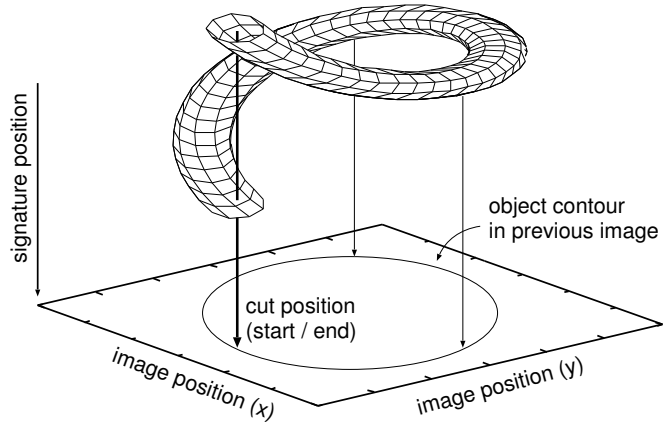


Figure 11.21: *Instead of considering the complete 3-D graph (2-D image position + signature position), only the nodes near to the original contour are used in the computation. The center of the spiral corresponds to the previous contour.*

Graph pruning

Carrying out a shortest circular-path search on the three-dimensional graph as constructed above is intractable, since this graph would consist of $W \cdot H \cdot C$ nodes, where $W \times H$ is the image size and C is the object contour length. However, if we assume that the object motion is limited, we can use the approach of Section 11.4.1 to restrict the search to a small corridor around the previous contour. More clearly, this means that all nodes (x, y, z) that are more than d pixels away from the previous object contour are removed from the graph. The resulting graph looks like the two-dimensional graph of the Corridor Scissors algorithm, but every node is duplicated C times in the z-direction. This can be visualized as a hollow tube, extruded from the object contour in the z-direction.

Even though this graph is already significantly smaller, we can reduce the size further by limiting the maximum amount of shrinking and stretching of the contour. Let $C_i = (x_i, y_i)$ be the sequence of pixels in the original contour. Assuming that the new contour has to follow the old contour with not more stretching or shrinking than an offset of s pixels along the contour, we can also exclude nodes $v_{x,y,z}$ from the graph, if the z-coordinate of the node deviates more than s from the previous object contour. When

combined, we obtain the following set of remaining nodes.

$$V = \{v_{x,y,z} \mid \exists (x_i, y_i) : \underbrace{|x_i - x| < d \ \wedge \ |y_i - y| < d}_{\text{limited motion}} \quad \wedge \quad \underbrace{|i - z| < s}_{\text{limited deformation}} \quad \}. \tag{11.4}$$

The resulting graph has the shape of a spiral with one turn around the object (Fig. 11.21). The number of nodes in this graph is approximately Cd^2 only.

Fast approximate calculation of shortest circular paths

The graph that we obtain for the object-signature fitting is non-planar, and we cannot use the algorithm described previously for computing the shortest circular path. However, for this graph it also holds that the length of the circular path is large compared to the corridor width. Previously, we have observed that for these long corridor graphs, there usually exists a subpath that is common to all shortest paths along the corridor. If we simply assume that such a common subpath exists, we can define a fast algorithm for shortest circular paths.

The algorithm is motivated by the AP1 algorithm for planar graphs. We first conduct an ordinary shortest-path search from the "left" side of the corridor graph to the "right" side, similar to Step 2 of AP1. Again, we mark the last possible node at which the shortest-path tree to the destination nodes is rooted as v_r. In a second step, we conduct a similar shortest-path search, but now in the opposite direction, starting from the just found node v_r to the "left" side. The node at which the paths to the destination nodes split is marked with v_l. It only remains to connect v_r with v_l in a similar way as described in Step 4 of the algorithm for planar graphs.

Assuming that a common subpath exists, this algorithm computes the correct shortest circular path. Otherwise, the correct solution might not be found, but it is still nearly optimal, as it is composed of two shortest-path segments.

11.4.4 Tracking results

This section, presents tracking results for three sequences. The first sequence shows a car that is seen in different views throughout the sequence, while in later parts of the sequence, the car passes behind some trees. The object contour was initialized at the first frame with the normal Corridor Scissors algorithm. This contour was used to derive the object signature, and this signature was kept fixed through the complete tracking (Fig. 11.22). We see that the tracked contour follows the object. It is interesting to notice that the tracked contour keeps locked at the parts of

the car that were defined in the first frame. In later frames, when the view of the car changes, new parts become visible, but the tracked contour still follows the originally defined content. This can be observed clearly at the rear of the car, but also at the back window, which was not visible in the first frame. The front window, which was present in the first frame, is occluded in later frames. To compensate for this, the algorithm pushes the contour away from the car to follow an image texture that is equally dark as the front window in the first frame. Around frame 40, the tracking result becomes worse and should be reinitialized or corrected.

In Figure 11.23, a new contour was initialized at frame 50 of the same sequence as above. During this part of the sequence, the car passes behind some trees. Since the signature holds information about the correct contour, the algorithm is not distracted when the car crosses the first tree (frame 70). However, in later frames (75-90), also the lighting conditions change and the algorithm loses the object.

Figure 11.24 shows tracking results for the *foreman* and the *paris* sequence. In the foreman sequence, it is interesting to see that the tracked contour at the helmet sometimes deviates from the true boundary. The reason is that the signature not only includes texture from the foreground, but also from the background. Hence, because a dark part of the background was visible along the top of the helmet in the first frame, the tracking algorithm searches for similar backgrounds in later frames. This leads to the tracking error that is visible in frame 100. On the other hand, at the left side of the helmet in frame 100, the contour is inside of the helmet, because the algorithm tries to prevent to pass by the dark area in the background.

In the *paris* sequence, the woman is first tracked successfully, but then, the algorithm locks to a local optimum around frame 100, because of fast object motion. However, in a later frame, the algorithm recovers again and yields the correct contour.

11.5 Summary and conclusions

This chapter has introduced Corridor Scissors as a new technique for semi-automatic image segmentation. This tool provides an elegant user interface, in which the object is marked coarsely with a broad corridor around the object border. Within this corridor, the object border is detected with pixel accuracy. Since the path within the corridor is adapted each time the corridor shape is changed, the interface supports incremental modifications to the segmentation to improve its quality.

The Corridor Scissors tool is based on a newly developed algorithm for computing shortest circular paths, which was also described in this

chapter. In practical cases, the described algorithm has the same computational complexity as an ordinary shortest-path computation using the Dijkstra algorithm. The described algorithm is applicable to general planar graphs and in an approximation also for some types of non-planar graphs. Consequently, our algorithm is also useful for other applications like shape-matching [177] or crack detection in borehole core images [7], in which comparable shortest circular-path problems occur.

Because of the low computation time of the shortest circular-path algorithm and the fact that the algorithm only has to consider the pixels within the corridor, the user-interface of the Corridor Scissors tool operates in real-time. Compared to the Intelligent Scissors algorithm, which computes the shortest paths to all pixels, this provides a significant *a-priori* reduction of the data to be processed.

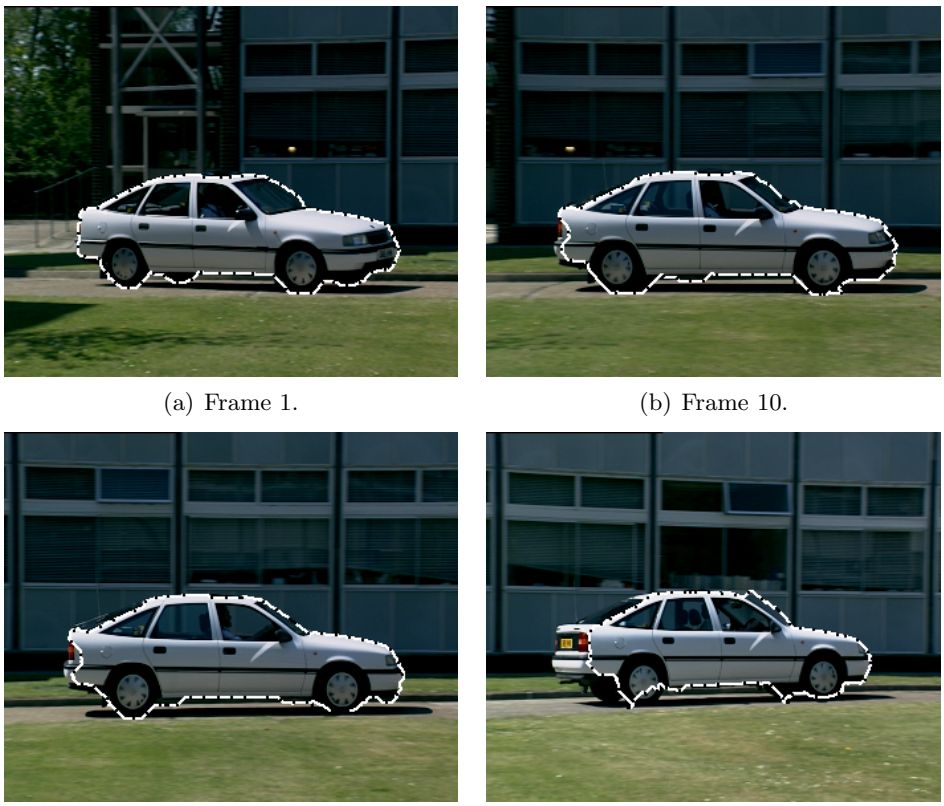
Finally, the Corridor Scissors tool was extended with a tracking algorithm. This tracking algorithm saves the texture around the object outline in a signature vector, which it is using during the tracking process to increase the robustness of finding the correct object border and decrease the probability of being irritated by background clutter. The tracking algorithm is also based on the shortest circular-path algorithm, but operating on a different cost function, which incorporates the signature information.

11.6 Discussion on the signature tracking technique

The concept of integrating knowledge about the object texture in the tracking process appears promising, since it improved the tracking results. Note that the object signature used in the tracking can also be considered as some kind of object model. However, we have observed two problems with this model that should be considered in future work.

First, the texture blocks in the signature vector include both foreground content and background content. In many cases (see *foreman* sequence), this leads to wrong contours because the algorithm finds a mismatch in the background texture. We conducted experiments to exclude the background content in the comparison. This is easily possible, as the foreground object mask is available from the previous segmentation. However, this resulted often in the situation that the detected contour moved inside of the object if it has a uniform color.

The second observation is that the tracking results can be improved by including also shape information. It is our opinion that this will improve the robustness against being trapped in a locally optimum contour. Furthermore, this may also help to solve the first problem. To see this, assume

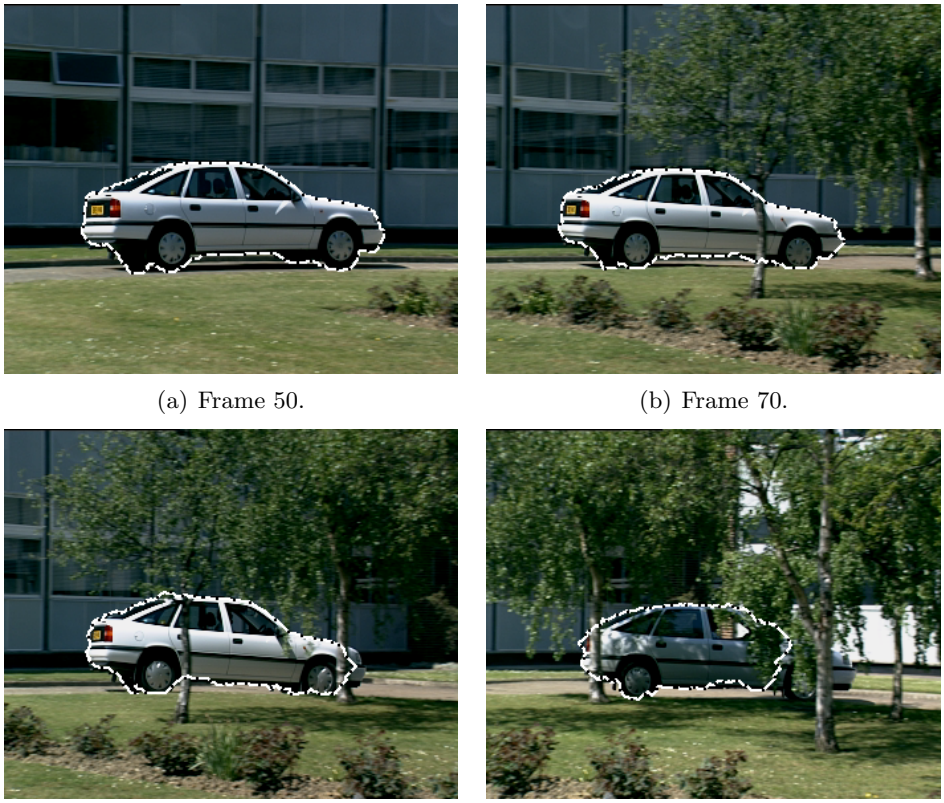


(a) Frame 1. (b) Frame 10.

(c) Frame 20. (d) Frame 40.

Figure 11.22: *Tracking in the* vectra *sequence. The signature is initialized in frame 1 and kept fixed throughout the tracking. Because the 3-D view of the car changes but the signature still contains the object outline from the original view, the computed outline differs from the true object boundary. Note especially the contour changes at the rear of the car and the front window.*

again that we would exclude background texture from the computation of matching cost. To prevent the contour to move inside the object, we have to apply some *force* to push the contour outwards. This force could be realized by the *a-priori* shape information. Unfortunately, object shape is a global feature that is difficult to integrate in the proposed framework. One promising direction may be to replace the object contour model with a mesh-based object surface model [21, 44], because this includes more texture information from the object and a shape-deformation cost can be defined based on the mesh geometry.



(a) Frame 50. (b) Frame 70.

(c) Frame 75. (d) Frame 90.

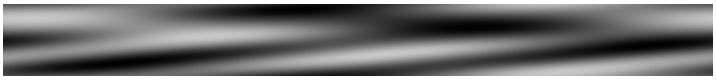
Figure 11.23: *Tracking the car while it passes behind some trees. The tracking is reinitialized at frame 50 of the sequence (a). When the car passes the first tree trunk (b), the texture information in the signature prevents the tracking from locking to the trunk as the new boundary. Later, in (c) and (d), large occlusions and changes in the lighting let the tracking deteriorate.*

(a) Frame 1. (b) Frame 1.

(c) Frame 100. (d) Frame 100.

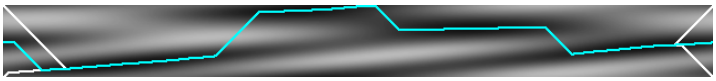
(e) Frame 225. (f) Frame 300.

Figure 11.24: *Tracking results for the* foreman *sequence (left column) and the* paris *sequence (right column).*



(a) Input image that defines edge costs.

(b) After Step 3 of AP1, a long common subpath is found.



(c) The shortest circular path.

Figure 11.25: *An example for which the shortest circular path can be computed with only AP1. Darker colors indicate lower cost.*

11.7 Appendix: step-by-step examples

This section illustrates the execution of the proposed shortest circular-path algorithm (Section 11.3.2) by some example runs. All examples use rectangular grid graphs with edges places as shown in Fig. 11.18. The costs are taken from an input image such that lower luminances correspond to lower costs. Circular paths are searched for along the horizontal direction through the rectangle (right column connects to left column). The selection of the examples does not reflect the typical input in the Corridor Scissors algorithm, but the examples are selected to clearly illustrate the execution of the algorithm. From the computational point of view, all examples are more complex than the typical input in the Corridor Scissors case.

Example 1 (Fig. 11.25)

The first example (Fig. 11.25) uses a low-frequency cost image. In Step 2 and 3, the algorithm computes the minimum-cost paths from the top-left corner to the top-right corner and, similarly, from the bottom-left corner to the bottom-right corner. These two paths share a long subpath along most of the corridor graph. Hence, AP1 can already determine the shortest circular path by connecting the nodes v_r and v_l in Step 4. Note that this step only computes a shortest path in the area composed of the two small triangles at both sides, since the large black areas can be excluded from the computation. This is the typical case as it usually occurs in the Corridor Scissors algorithm (see also Fig. 11.26).

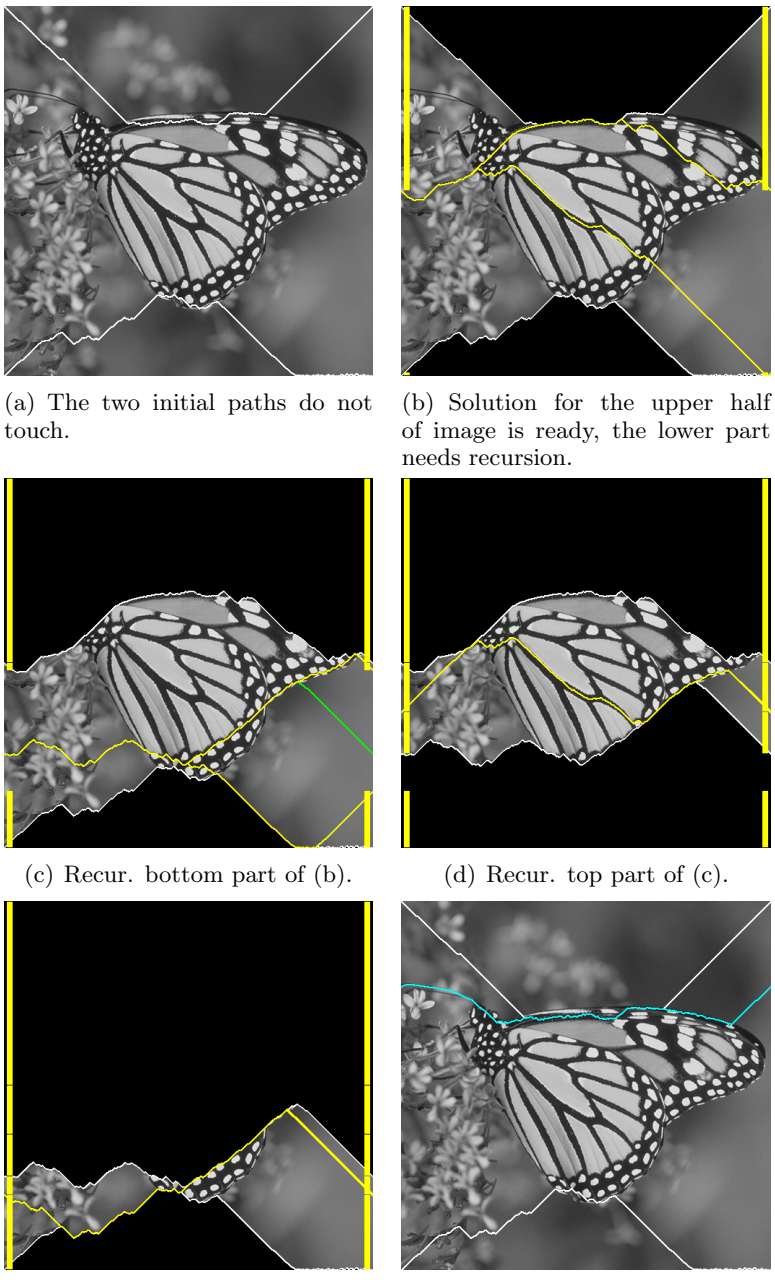
Figure 11.26: *The shortest-paths trees on both sides of a cut (compare to Fig. 11.11(c)) for a typical real-world example. It is clearly visible that all paths join quickly to a common path along most of the corridor.*

Example 2 (Fig. 11.27)

In a second experiment, we used a natural image as cost data (but note that the shortest circular path does not have any semantic meaning here). The algorithm starts with computing the two shortest paths along the top and the bottom side of the rectangle (Fig. 11.27(a)). Unlike in the last example, these two paths do not join, so that we cannot use AP1 compute the shortest circular path. Hence, we have to use AP2 for this example.

This part of the algorithm starts with the initial recursion step, in which a shortest-path search is initiated from the middle position on the left side. Indicated in Fig. 11.27(b) are the two last paths, starting from the middle position, that still touch the top and bottom paths from the last step, respectively. For the top path, the last position for which we still get a touching path, is the middle path itself. Consequently, we have the situation of two touching paths in the top half of the image and the shortest circular path in this area can be computed in one step (again by joining v_r and v_l). The range of $\bar{V}$ that is completed is indicated by the bars at the sides of the image. Unfortunately, the range that could be processed in the bottom half is very small and further recursions are required.

Since the top part of the image could be completed in one step, we only have to consider the bottom part in the recursion (Fig. 11.27(c)). In the remaining area, a shortest-path search is initiated again from the middle position at the left side. Note that the graph on which the search is now carried out is much smaller, since the top half can be excluded (black area). In this step, only a small range at the top and a larger region at the bottom can be completed. Because none of the two halves are finished completely in this step, both halves are further processed recursively (top



(a) The two initial paths do not touch.

(b) Solution for the upper half of image is ready, the lower part needs recursion.

(c) Recur. bottom part of (b).

(d) Recur. top part of (c).

(e) Recur. bottom part of (c).

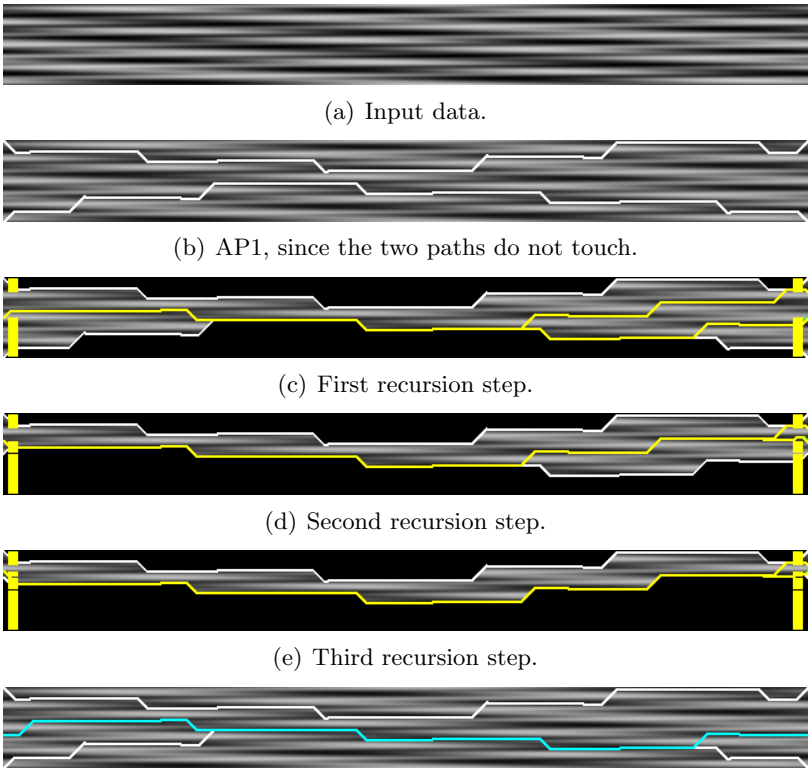
(f) Final result.

Figure 11.27: *Step-by-step example of the execution of the shortest circular-path algorithm. The luminance of the input image defines the cost. Black areas are parts of the graph that could are excluded from the computation.*

half: Fig. 11.27(d), bottom half: Fig. 11.27(e)). After these steps, all nodes in $\bar{V}$ are covered. The globally shortest circular path is selected as the shortest path of the solution from each of the processed ranges. This solution is presented in Fig. 11.27(f).

Example 3 (Fig. 11.28)

The input data that was synthesized for this example (Fig. 11.28) was motivated by the worst-case data (Fig. 11.16(a)) for the algorithm. The input shows many thin low-cost paths, so that there is no obvious common shortest subpath. Nevertheless, the two bounding paths (Fig. 11.28(b)) almost touch, and the shortest circular-path problem can be solved in only four steps of recursion. We made similar observations with input data that is just random noise. This suggests that the computation time is usually low, even with complicated input data. An important factor is the ratio of corridor length to its width, since the probability of joining paths also increases when this ratio increases.



(a) Input data.

(b) AP1, since the two paths do not touch.

(c) First recursion step.

(d) Second recursion step.

(e) Third recursion step.

(f) Result after fourth recursion step.

Figure 11.28: *An example case that is close to the worst-case pattern shown in Fig. 11.16. Here, AP2 has to be applied, but the computation can be completed with only four recursion steps.*