Chapter 3

A new Algorithm: Tracking Regions

Consider a system for a mobile robot, perceiving, reasoning and acting in real-time. Such a system might be divided into many parts, for instance vision, behavior, motor control, reasoning and communication. However, all parts share a common problem when performing calculation over time: Should all calculations be made from scratch or should the results of recent calculations be adapted to the current situation?

For instance, when the robot has planned a path and a moment later an object has moved, blocking the planned path, should the path planner calculate a completely new path or should it try to adjust the old path? A system able to adapt calculations instead of calculating everything from scratch takes advantage of the fact that the solutions to the current problem will smoothly vary in time. A frequent problem is, that robots create unsteady, oscillating data structures and plans. Moreover, the overall solution will typically be more efficient, since it uses the results of recent computations.

In the field of computational geometry, a new area of research has emerged due to this consideration. It is called *kinetic data structures* and it was introduced by Leonidas J. Guibas. For a review see [36]. Here, the question is how structures like the Voronoi-diagram, for instance, can be computed in a scenario where the underlying situation is continuously changing. The goal is to adapt an existing solution to the new scenario and not to compute the solution from scratch.

In real-time computer vision, the same problem exists. It would be wasteful to process the images independent of one another and to run image processing techniques each time from scratch.

Essentially, use of techniques like the system dynamics approach, which was described

at the beginning of chapter 2, exploits the continuity of the world underlying the images. However, one weakness of the method is that it needs a model of the environment. It would not be a weakness if methods were available that could automatically create the models themselves, but this would require methods that can start without a user-defined model.

In RoboCup, if a model and the initial estimate of the robot's pose are available, the location and appearance of the field lines in the image can be predicted. This allows to access only a fraction of the image data. On the other hand, as described in the previous section, there exist methods that can detect the field lines without the initial knowledge of the robot's pose, but at the cost of accessing more image data.

It would be nice to have a method that combined the advantages of both, accessing only a small fraction of the data without the need of a user defined-model and without having to know the robot position and distance calibration. In this section, we will develop such an algorithm, the complete C++ source code of which is enclosed in appendix B.

In order to develop the method, an important characteristic of natural images has to be exploited, the tendency of regions to contain similar colors or textures. Natural images do not look like figure 3.1a) but instead contain regions of similar color (figure 3.1b)). Particularly in the RoboCup domain, the images contain large green regions caused by

(a) (b)

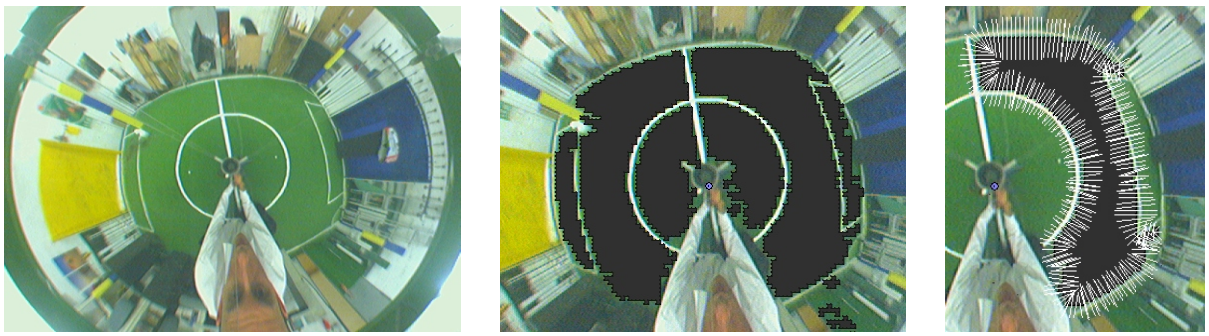
Figure 3.1: (a) This image only contains noisy points and does not exhibit regions of similar color. (b) As a contrast, a natural image usually contains homogeneous regions.

the green carpet and we will use these regions to detect the field lines.

The idea is to determine the boundary shape of the regions and to apply a local detector for the field lines around the boundaries (see figure 3.2). Thus, we use the boundaries as an index structure for feature searching in the image. This procedure is similar to that of using a model in that we limit the search space, but the method will be more flexible, since the geometry of the boundaries solely emerges from the content of the image.

Compared with a model, additional computation is necessary to determine the regions and their boundaries. However, we will see that homogenous regions have a nice property that can be exploited to develop an efficient method for that purpose: When working with a video stream, successive images are similar in most cases and when a region changes position from one frame to the next, these regions overlap in many cases.

This observation can be exploited in a way that the results from processing a frame are used to speed up the treatment of the next frame.



(a) (b) (c)

Figure 3.2: (a) The omni-directional image of a robot in the middle of the soccer playing field. (b) The regions between the lines as obtained by the tracking algorithm. (c) One region as an example: A detector is applied along lines perpendicular to its boundary.

The algorithm is based on the region growing paradigm [14][91][74][2][42][65]. However, in the following section, we extend the paradigm to track regions over time.

3.1 Extending the Region Growing Paradigm

In this section, we extend the region growing paradigm to track regions over time. Therefore, we first give a short review of a specific region growing algorithm that will be extended

later.

3.1.1 Region Growing by Pixel Aggregation

In *Region Growing by Pixel Aggregation* one starts with a small region (i.e. a single pixel) and consecutively adjoins pixels of the region's neighborhood as long as some homogeneity criterion is fulfilled. This method can be implemented in the following way: We reserve a two-dimensional array V that has the same size as the image and that has a boolean entry for each pixel that indicates whether the respective pixel has been visited yet. Further, we maintain a queue Q that stores the spreading boundary pixels (their coordinates) of the region during the execution of the algorithm. We refer to the stored elements in Q as drops following the idea that we pour out a glass of water at the seed pixel and that the drops of this water spread over the region. Q stores the boundary drops at each time during the execution of the algorithm. Initially, if we start with a single pixel, Q stores a single drop corresponding to the pixel that is also marked as visited in V. The algorithm continues with a loop that is performed as long as any drops are stored in Q. In each pass of the loop, one drop is extracted from the queue and the neighboring pixels are investigated. In the case of a four-connected neighborhood, we inspect the top, right, bottom and left pixels. After insuring that the pixels have not yet been visited, we determine whether they hold for a specific homogeneity criterion, color similarity for instance. For each pixel that conforms to this condition, a new drop is instantiated and stored in Q. After the loop terminates, Q is empty and the region is marked in V. Figure 3.3 shows an example of a growing process that finds the region of the yellow goal.

3.1.2 The Key Observation

Our goal is to efficiently track regions over time. To give a specific example, we want to track the yellow goal while the robot moves. Note, that later we will not track the yellow goal, but the green regions. However, the algorithm can more easily be explained with the example of the yellow goal. Assume that the robot starts at pose P_A and takes an image I_A. Then the robot moves a little to pose P_B and takes another image I_B. Assume further that we have determined the region A of the yellow goal in I_A and the corresponding region B in I_B by the above region growing algorithm. If the video frequency is high enough with respect to the movement of the robot, the two regions will overlap as depicted in

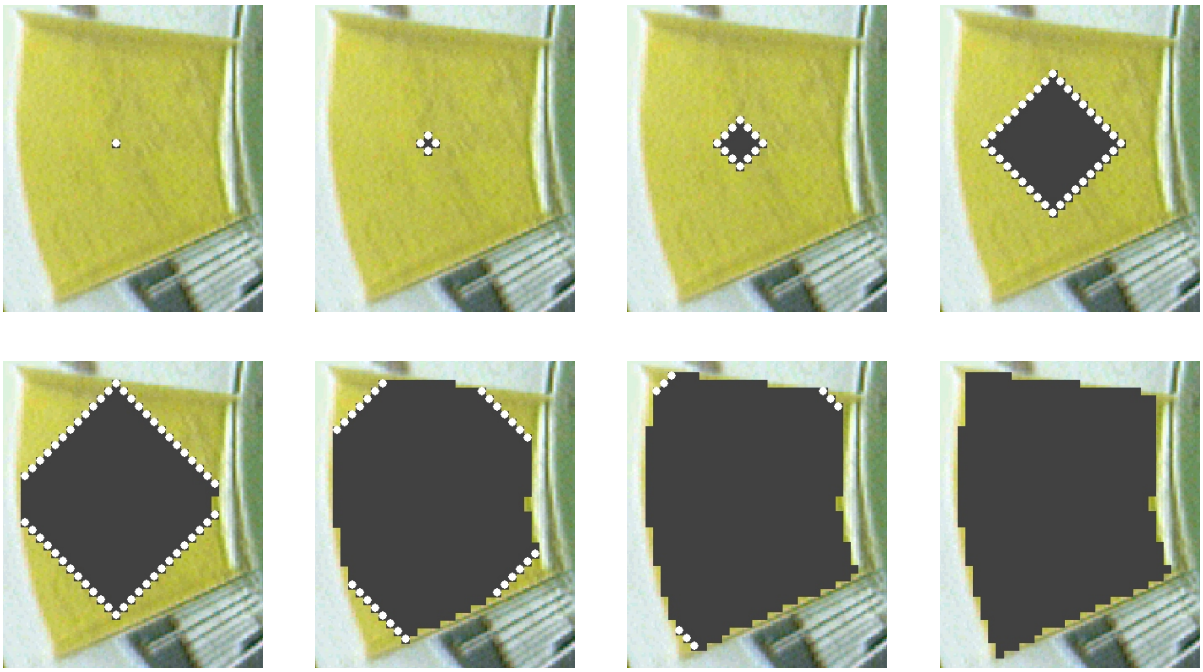


Figure 3.3: The array V is visualized by the dark gray surface (visited elements). The white circles represent the drops. Here, the algorithm does not work on single pixels, but on blocks of 4×4 pixels.

figure 3.4. If we apply the region growing algorithm separately to image A and B, the

(a) (b) (c)

Figure 3.4: The regions of the yellow goal are determined (a) before and (b) after the robot has moved. (c) The two regions overlap.

running time for each is linear with respect to the number of pixels (or blocks of pixels) within the respective regions.

However, we want to develop a more efficient method. Assuming that we have extracted region A, then roughly speaking, we want to use the drops of A's region growing to somehow flow to the region of B by making use of the overlap. The drops of region A should first shrink to the intersection S of A and B and then grow to the boundary of B. Thus, in order to find region B, we don't start with a seed pixel, but with a whole seed region, the intersection of A and B. Figure 3.5 sketches this idea. To realize the algo-

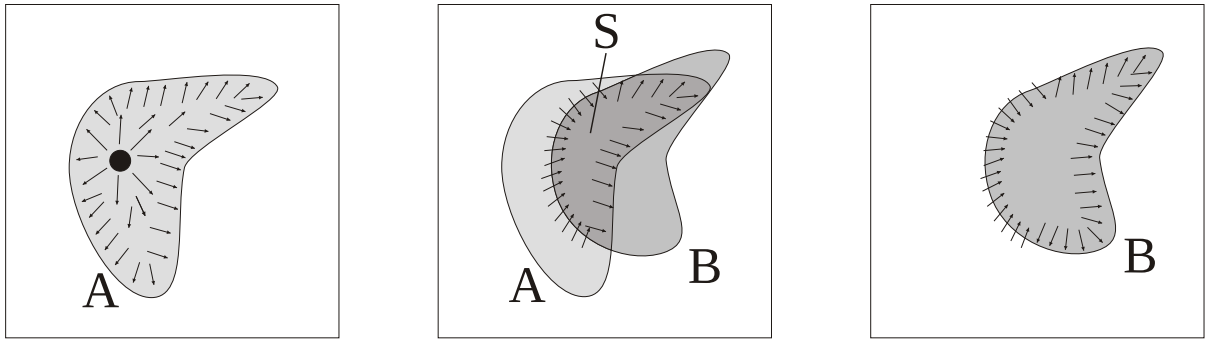


Figure 3.5: (a) Seed pixel and initial growing to the boundary of A. (b) Shrinking to the intersection of A and B. (c) Growing to the boundary of B.

rithm, a method to shrink region A to the intersection of A and B has to be developed. This will be done in the following subsection.

3.1.3 Shrinking Regions

We will develop the shrinking method in two steps. First, we consider the case of shrinking a region without stopping. That is, the region shrinks until it vanishes. Next, we modify the method so that shrinking stops at the intersection of A and B.

In the first step, we don't need any image information, just the array V in which region A is marked, and a queue of drops at the boundary of that region. As we will need two different queues for growing and shrinking later, we denote the queue used here as Q' to avoid confusion. We apply the same operations as in region growing with one exception: We reverse the meaning of V. As a result, the drops can only spread within the region marked in V and since they are initially placed at the boundary, their only means of escape is to move from the outer to the inner part of the region. Instead of marking

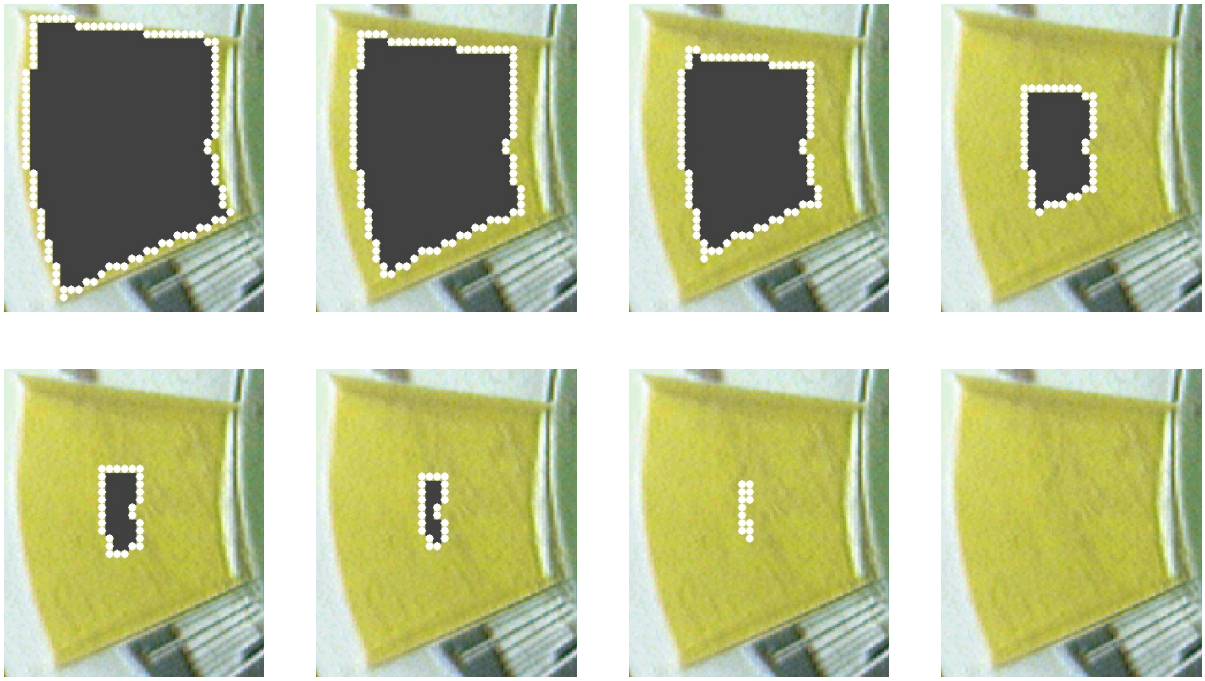


Figure 3.6: The region shrinks until it vanishes. No image information is required for this process.

elements in V as in region growing, entries in V are cleared while the drops spread. At the end, Q' is empty and V is cleared completely. This process is illustrated in figure 3.6.

In the second step, we use image I_B to determine when shrinking should stop. We emphasize that the initial region (marked in V) comes from image I_A while image I_B is referenced during the method. Each time a drop has been extracted from Q', we verify the homogeneity criterion for the corresponding pixel in image I_B. If the pixel belongs to the region, the drop is no longer allowed to spread. As a result, the region shrinks to the intersection of region A and B as depicted in figure 3.7. This is exactly inverse to the growing, where drops only spread to neighbors that fulfill the homogeneity criterion.

3.1.4 Alternating Shrinking and Growing

We want to alternate shrinking and growing in order to track regions. However, some problems concerning the interface between the two processes must be solved. The first problem is that after growing, the queue of drops is empty, but shrinking initially requires

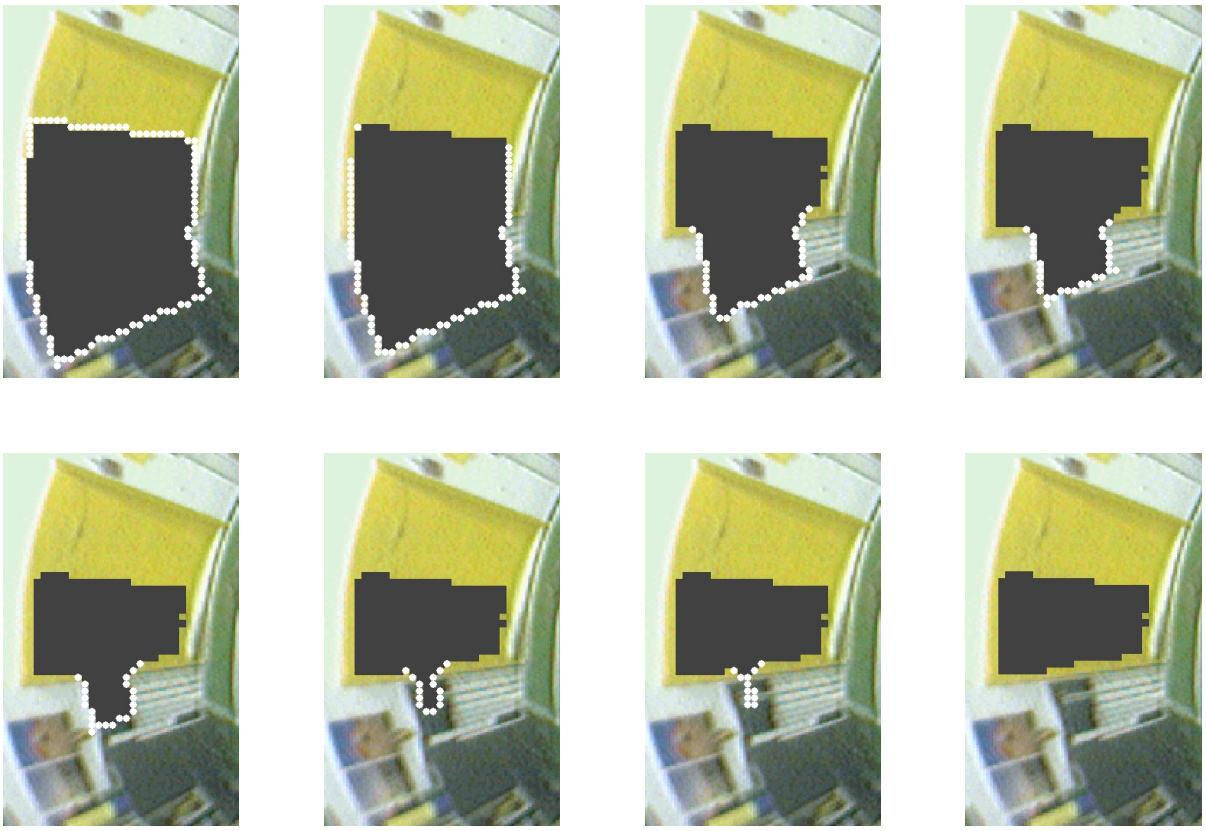


Figure 3.7: The region shrinks to the intersection area, that is, until all drops are within the region of the new image.

a list of drops at the boundary of the region. In the same way, shrinking ends with an empty list of drops but growing requires seed drops. To solve this problem, each of the two processes, growing and shrinking, has to build the initial queue for the other procedure. We accomplish this by using two queues, Q and Q'. Growing assumes the initial drops to be in Q and after execution Q is empty and Q' has been built up, which stores the initial drops for shrinking. Shrinking runs with these drops and initializes Q for the next growing.

When the neighbors of an extracted drop are inspected during growing, the drop is inserted into Q' as an initial drop for shrinking if any of its neighbors do not belong to the region. Vice versa, a drop that is extracted from Q' during shrinking is inserted to Q as an initial drop for growing if shrinking stops for this drop. A complete listing of the source code in C++ is available in Appendix B. Here, we give the pseudo code for the growing and

shrinking procedure:

Procedure 1 *Grow()*

Input:	*array V*	*marks whether a pixel has been visited yet*
	queue Q (FIFO)	*maintains the drops of the growing process*
	image Ia	
Output:	*queue Q'(FIFO)*	*the initial drops for shrinking*

```
WHILE (NOT Q.IsEmpty()) BEGIN
    d:=Q.ExtractDrop();
    atBorder:=false; //a temporary boolean variable
    FOR (all neighbors b of d) BEGIN
        IF (b fulfills the homogeneity criterion) BEGIN
            IF (V is not marked at the position of b) BEGIN
                Mark V at the position of b;
                Q.Append(b);
            ELSE atBorder:=true
        END
    END
    IF(atBorder) Q'.Append(d);
END
```

Procedure 2 *Shrink()*

Input:	*array V*	*marks whether a pixel has been visited yet*
	queue Q' (FIFO)	*maintains the drops of the shrinking process*
	image Ib	*the next image*
Output:	*queue Q(FIFO)*	*the initial drops for growing*

```
WHILE (NOT Q'.IsEmpty()) BEGIN
    d:=Q'.ExtractDrop();
    IF (the pixel in Ib corresponding to d does not fulfill the homogeneity criterion) BEGIN
        FOR (all neighbors b of d) BEGIN
            IF (V is marked at the position of b) BEGIN
                Clear V at the position of b;
                Q.Append(b);
            END
        END
    ELSE Q'.Append(d);
END
```

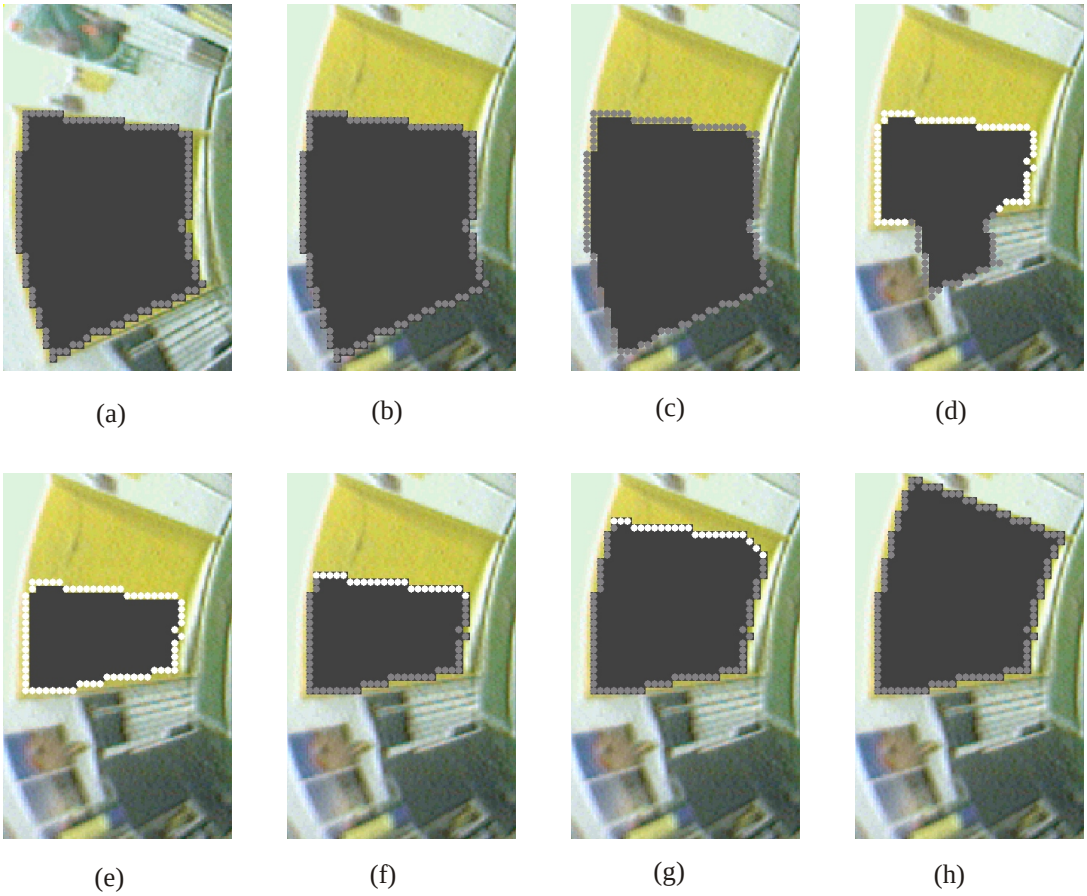


Figure 3.8: The drops of Q and Q' are painted white and gray respectively throughout the following steps: (a) Image I_A after growing. (b) The new image I_B (c) After clearing V at the drops of Q'. (d) During shrinking. (e) After shrinking. (f) Start of growing. (g) During growing. (h) After growing.

In order to start the shrinking process after the growing process, one last supplementation has to be done. Between growing and shrinking, elements in V which correspond to the positions of the initial drops for shrinking in Q' have to be cleared. This is because during shrinking the drops should always be on unmarked fields of V.

Figure 3.8 illustrates the combination of growing and shrinking and shows how a region is tracked over two consecutive images. The drops of both queues, Q (white) and Q'(gray), are depicted. By consecutively alternating shrinking and growing, a region can be tracked through a sequence of frames.

3.1.5 Applicability

Consider an image with only one region being tracked. Assume that outside the region there is an object that moves very quickly towards and into the region. As a result, the object will be within the region in the next image. Thus, the object is an island within the region. This island will not be detected by the algorithm, since due to the fast movement, the object never penetrates the boundary of the region but "jumps" over it from frame to frame. As a consequence, the drops will not detect the object in the shrinking phase, since the drops are never next to the object. Therefore, the algorithm should not be used to track small and fast regions. Here "small and fast" means that the regions, which emerge from the object, do not overlap in two consecutive frames. Interestingly, a high frame rate supports the applicability of the algorithm, since regions will more likely overlap.

3.1.6 Running Time

The efficiency of the algorithm is caused by two facts: First, a region can be tracked without touching pixels not concerned with the region. This is possible because the drops act as indices to the image and the method is able to update the indices without accessing image information outside the regions. Second, the computational load of tracking the region is only a fraction of the load required to process the entire region, because only the non-overlapping parts are processed.

In the following, we investigate the running time more precisely. When tracking the region over the two images I_A and I_B, the running time of the algorithm is primarily determined by the number of drops that are inserted and extracted from the lists, and the number of memory accesses to the image and array V. However, the number of memory accesses to the image and array V is up to a constant equal to the number of drop insertions. Three processes contribute to the running time:

Clearing Here the elements in V that correspond to the drops in Q' after growing are cleared. Thus, the running time is $O(k_A)$, where k_A is the number of drops at the boundary of region A.

Shrinking The main loop within the shrinking step (procedure 2) executes as long as drops are extracted from queue Q'. Initially, there are k_A drops in the queue. During execution, drops are also inserted into Q' and since Q' is empty at the end, they

also contribute to the number of main loop executions. The number of drops is equal to the size of the shrinking area, which is the size of region A without the part overlapping with region B. We denote this number with $n_{A\setminus B}$. Furthermore, the list Q is built by inserting $k_{A\cap B}$ drops, the number of drops at the boundary of the intersection area of region A and B. Therefore, the running time of shrinking is $O(k_A + n_{A\setminus B} + k_{A\cap B})$.

Growing Analogously, the running time of growing is $O(k_{A\cap B} + n_{B\setminus A} + k_B)$.

Thus, the total running time is

$$O(k_A) + O(k_A + n_{A\setminus B} + k_{A\cap B}) + O(k_{A\cap B} + n_{B\setminus A} + n_{B\setminus A} + k_B) = O(k_A + k_B + k_{A\cap B} + n_{A\setminus B} + n_{B\setminus A}) \quad (3.1)$$

Hence, the running time depends on two types of terms: In the first type, $(k_A, k_B, k_{A\cap B})$ the contributions depend on the length of the region boundaries. In the second type, $(n_{A\setminus B}, n_{B\setminus A})$ the contributions are proportional to the non-overlapping parts of the regions A and B (see figure 3.4c), which is a fraction of the size of the entire regions (in the worst case equal to the size of the regions).

3.1.7 Controlling the Tracking

When regions of two successive images do not overlap, there is no intersection area and the shrinking step will delete the region. Hence, if the movement between two consecutive images is too large, a control mechanism has to be integrated into the loop of shrinking and growing. The control mechanism checks if tracking has lost the region. In this case, an initial search has to be started. Depending on the application, this procedure might search within the whole or a part of the image for a pixel or a block of pixels having a certain color or texture that satisfies the homogeneity criterion and maybe some other detection criterion. Figure 3.9 sketches how the controlling mechanism is integrated into the algorithm.

Also, regions can be discarded from tracking. In this way, the computational power can be concentrated on regions of interest. To exclude a region from being tracked, one simply has to delete its entries in V and the corresponding queues. This can be accomplished by shrinking without a stopping criterion as illustrated in figure 3.6.

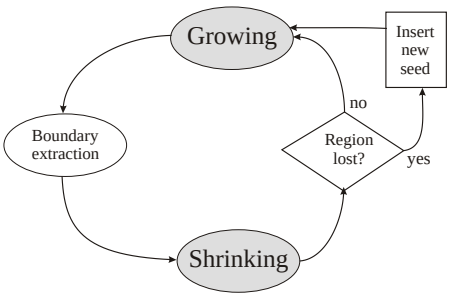


Figure 3.9: The controlling mechanism detects when a region gets lost. In this case, an initial search is started, re-initializing the tracker with new seeds. Shape information is retrieved by boundary extraction after the growing phase.

3.1.8 Homogeneity Criterion

The homogeneity criterion specifies whether two neighboring pixels or blocks of pixels belong to the same region. There are many possible ways of defining such a criterion. It can be based on lightness, color or texture. It can be formulated either relatively by specifying, for instance, that two pixels are homogeneous if the difference in intensity is below a certain threshold, or absolutely, by classifying each pixel individually and comparing whether their classes are equal. An example is separation into color classes. In our RoboCup application, for instance, we classify the membership of each pixel (block of 4×4 pixels) in the color classes green, white, yellow, blue, orange and black and we define two blocks of pixels as homogeneous if they belong to the same color class.

3.1.9 Tracking Several Regions

Extending the algorithm to track several regions, we assume that the homogeneity criterion is based on classification. We first consider the case that all tracked regions are of the same class. The green surfaces between the marking lines of the soccer playing field depicted in figure 3.2 represent such a case. The algorithm can be applied without change. However, the initialization differs, since seed drops have to be placed into the list Q for each region. Thus, Q and Q' store the drops of several disjunct regions.

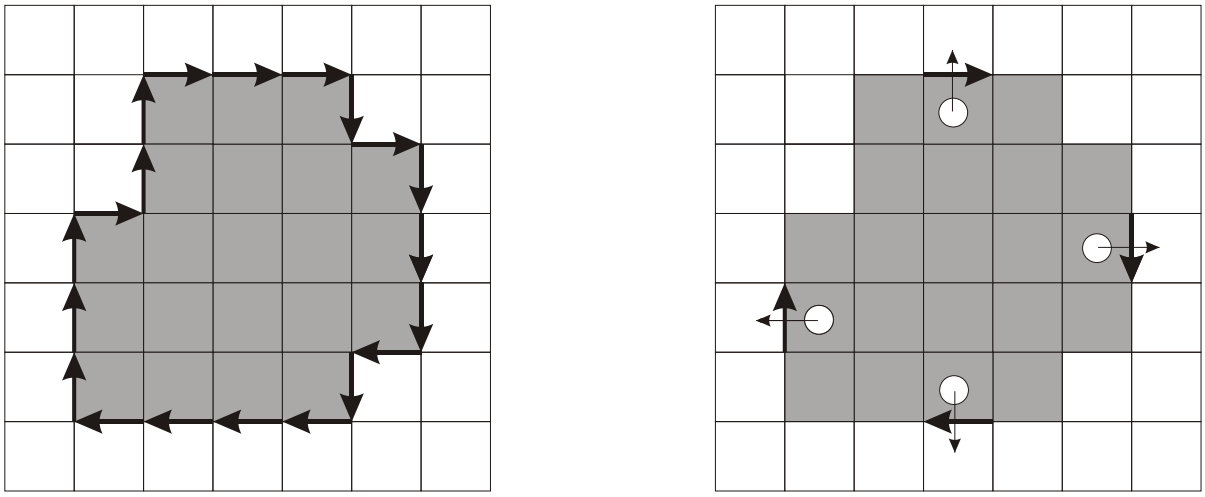
Next, we want to track several regions of different classes. One possible approach would be to use only the one pair of lists Q and Q' for all drops and to append the class information to each drop. But following this approach, difficulties arise when we want to extract the boundaries of the regions later, since the drops of different classes cannot be

accessed separately.

Therefore, we decided to use a separate pair of queues for each class and perform growing and shrinking for each pair individually. In contrast to the separate lists, all classes use the same array V whose elements now store the class information instead of boolean values. A special value is reserved to mark unvisited fields.

3.2 Boundary Extraction

The boundary of each region consists of a chain of small edge vectors (see figure 3.10(a)). Each edge vector represents one of the four sides of a pixel (or block of pixels) and the last edge vector ends at the beginning of the first.



(a) (b)

Figure 3.10: (a) The boundary of a region is composed of a sequence of edge vectors. (b) The directions of the edge vectors depend on the direction of the spreading when they are detected during the growing phase.

The extraction of the boundary incorporates two steps. First, we determine the edge vectors and represent them in a specific data structure, the *connectivity grid*. Second, we extract the boundary by following the sequence of edge vectors.

The first step is integrated in the growing phase. Remember that during growing, the drops spread until they reach the boundary of the region. Consider a drop that is next to the region boundary. When the drop spreads, its neighboring positions are investigated and since the drop is next to the boundary, at least one of its neighbors will not fulfill the homogeneity criterion. Each time such an event occurs, we represent the corresponding edge vector in the connectivity grid. The edge vectors can be considered as small line segments that separate the drops within the region from their neighboring positions outside the region. However, the line segment is directed, allowing us later to follow the boundary contour. Figure 3.10(b) illustrates the relationship between the edge

vector's direction and the direction from the drop within the region to the position outside the region.

The connectivity grid corresponds to the image grid, except that the cells of the grid do not correspond to the pixels, but to their corner points. Therefore, the width and height of the grid exceeds those of the image by one, respectively. We reserve four bits for each grid point, expressing in which of the four possible directions an edge vector points. In Figure 3.11a) a small region is shown. Figure 3.11b) illustrates the corresponding connectivity grid of its boundary contour. The four bits of each cell are depicted as quarters of a circle to demonstrate their relation to the direction of the edge vectors. A black quarter means that an edge vector is present. Figure 3.11c) shows the superposition of the image and the connectivity grid with the edge vectors also inserted. Figure 3.11d) demonstrates that two bits can be set at the same time when neighboring regions touch at a corner. This approach is similar to [14], but our method occupies less memory, since [14] employs a supergrid of size $(2n+1) \times (2m+1)$ and we use a grid of size $(n+1) \times (m+1)$, where $n \times m$ is the size of the image.

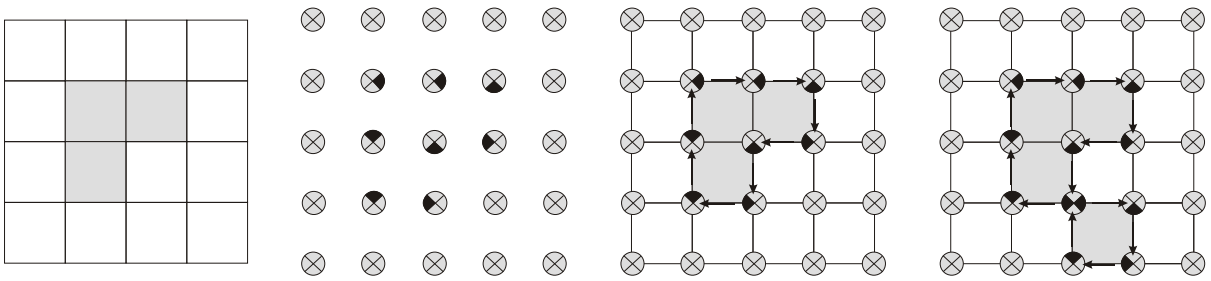


Figure 3.11: Representing regions and their boundaries: (a) Example image with a gray region; (b) The connectivity grid, whose points have four bits to indicate the direction of outgoing edges; (c) The grid superimposed to the image. The flags indicate the boundary contour. (d) Several flags within a grid point will be set, if different regions touch each other.

Boundary extraction can be accomplished by following the sequence of edge vectors. But since some grid points might offer several continuations, it is important to apply a rule in order to guarantee that each boundary is closed. One should always decide for the clockwise or counterclockwise possibility first. Depending on the direction of

rotation by which the edge vectors have been inserted into the grid, these rules yield the smallest/largest possible region boundary. Each time an edge vector is followed, we delete the edge vector from the grid. Hence, after all boundaries have been extracted, the grid is completely cleared.

As mentioned above, the insertion of edge vectors into the connectivity grid takes place during the growing phase. When a drop is detected to be at the boundary of a region, the corresponding edge vector is inserted. An initial grid point must be present, to follow the contour. Therefore, after growing, we use the first drop in Q' to obtain this starting point. Note, that the position of the drop itself is not the starting position, but one of the corresponding corner points represented in the grid.

More than one region and corresponding boundary contours might exist. Therefore, after having extracted the first contour, we scan Q' for the next starting point. While passing through the drops, we look at the connectivity grid whether an edge vector is present next to the actual drop. Four possible grid points have to be verified for each drop. The first edge vector found specifies the starting point of the next boundary contour. To avoid accessing drops in Q' several times while searching for the start positions of successive boundary contours, we use a pointer that points to the first drop in Q' that has to be inspected when we search for the starting point of the next contour. At the beginning, the pointer points to the first drop in Q'. With each boundary extracted, we move the pointer ahead. In this way, when searching for the next starting point, we can skip the elements in front of the pointer.

Unfortunately, moving the pointer ahead can not be carried out while extracting the contour, because there is no link between the grid points and the position of the corresponding drops in the queue Q'. Note, that even if only one region is tracked, the drops in Q' are not necessarily ordered along its contour. This becomes apparent in the case depicted in figure 3.12a)-c). Here, a region is shown that has an island, which means that within the region there is an area that does not satisfy the homogeneity criterion with respect to the region. The region has an outer and inner boundary contour. A seed drop is placed next to the outer boundary contour and growing is started. In each of the figures 3.12a)-c), the queues Q an Q' are shown at the top and bottom of the figure. Figure 3.12b) shows the state of the algorithm after the first loop of the growing procedure and figure 3.12c) shows the state after the second loop. Note that in 3.12c), the boundary drops in Q' are not stored in the order of the course of the contour. Indeed, the drops

even change between inner and outer boundaries.

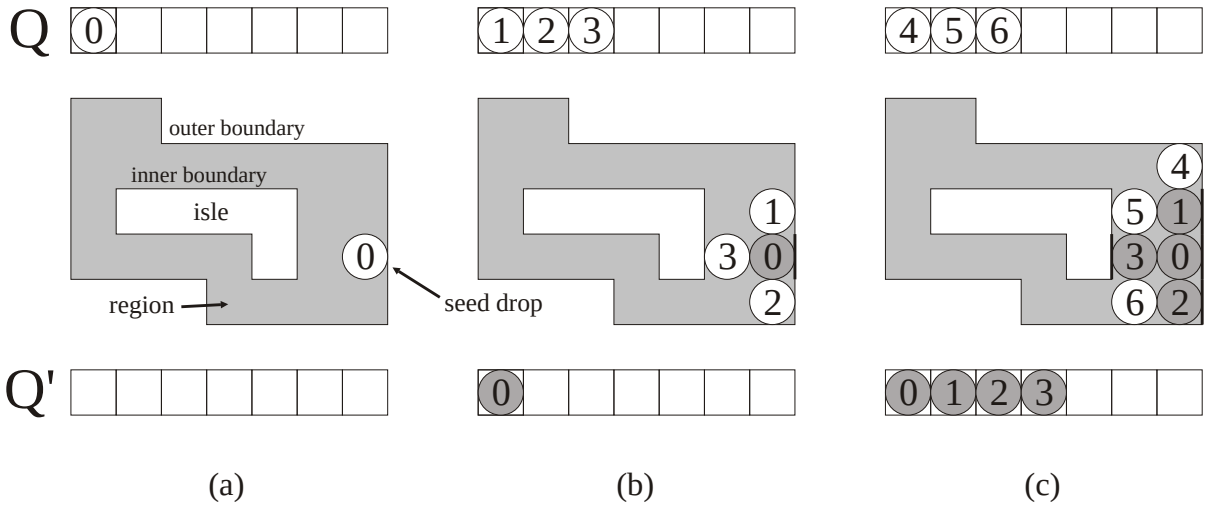


Figure 3.12: This figure illustrates that the order of the drops in Q' might not correspond to the course of the boundary contour. Even if only one region is tracked, there might be more than one boundary contour if the region has an island as depicted here. The queues Q and Q' are shown (a) after seeding, (b) after the first and (c) after the second loop of the growing procedure.

Thus, the pointer cannot be updated while extracting the boundary. Instead, we pass through Q' after boundary extraction and search for the first drop that is next to an existing edge vector. Remember that the edge vectors are cleared during extraction. Therefore, the found edge must be a starting point of a new contour and we move the pointer to the corresponding drop.

3.3 Extracting the Field Lines By Tracking Regions

Many vision systems rely on the detection of discontinuities in the image. We use the term "discontinuity" instead of "edges" to emphasize that edges are not the only features of interest. Lowe [73] uses SIFT (Scale Invariant Feature Transform) features for stereo matching, ego-motion estimation and map construction. Harris's 3D vision system DROID [40] uses image corner features for 3D reconstruction. In [76] edges are extracted for motion segmentation. However, in all these applications, the features are detected by accessing all pixels in each frame. Since the features represent inhomogeneities in the image, homogeneous regions will surround them. In many cases, these regions will be large enough to make the application of our region tracking algorithm meaningful. Thus, touching the whole image can be avoided by applying the respective detector at the boundary positions of the regions.

Moreover, more specialized detectors can be used. For instance, when searching for edges, a direction selective detector can be applied, because the normal direction of the boundary contour of the region can be determined. This can be done during the extraction of the contour. While following the edge vectors, the local tangent and thus, also the normal direction, can be calculated. However, since the boundary contour is a sequence of small edge vectors that have only four different perpendicular directions, several edge vectors in the neighborhood of the actual point should be used for tangent calculation. The simplest method is to use two constantly spaced points on the contour, which are iteratively moved one edge vector ahead. Then, the line connecting the two points can be considered as the tangent of the actual point (see figure 3.13 for illustration). Another

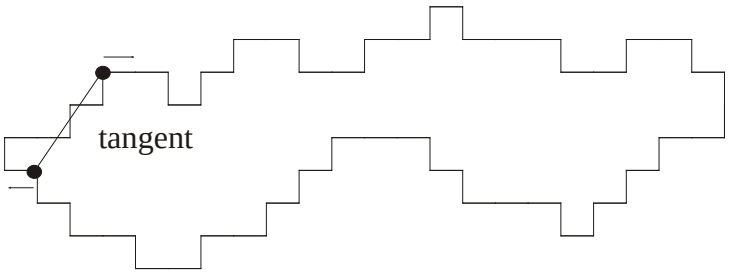
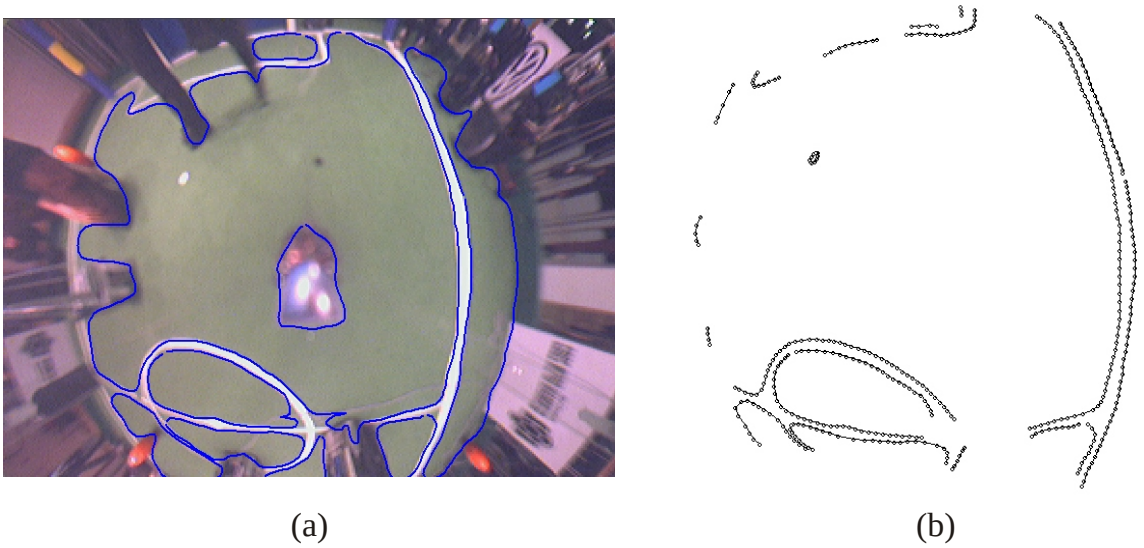


Figure 3.13: The tangent at a point on the contour can be calculated by connecting two points in the neighborhood of the actual point. To obtain the tangents of successive points, the two endpoints are shifted stepwise along the contour.

method to calculate the tangent directions is to smooth the boundary contour before computing the vectorial difference between successive points.

In our RoboCup application, we use this way of proceeding to detect the white field markings. We track the green regions of the playing field and apply a detector along line segments that are perpendicular to the contour. The detector searches for color transitions from green to white and back to green. In this way, we obtain sequences of points that represent the field lines as illustrated in figure 3.14b).



(a) (b)

Figure 3.14: (a) The boundaries of the tracked regions are smoothed and a local line detector is applied. (b) The resulting line contours consist of a set of lines where each line is represented by a sequence of points, which are marked by small circles.

Using this method, the field lines are represented twice, once from each of the two neighboring regions. At first glance, this seems to be a disadvantage, but we will see later that feature detection benefits from this fact.

Although the method works well, one consideration needs to be addressed. Two regions adjoin at each field line. The fact that they are only separated by relatively thin lines seemingly produces a problem. One might think that a similar effect as described on page 58, where an object intrudes into a region without being detected, can occur with the lines. In fact, if the robot moves fast, a line can intrude into the region without meeting a drop. This scenario is illustrated in figure 3.15.

Indeed, the line would get lost from tracking, if not for the drops at the intersection

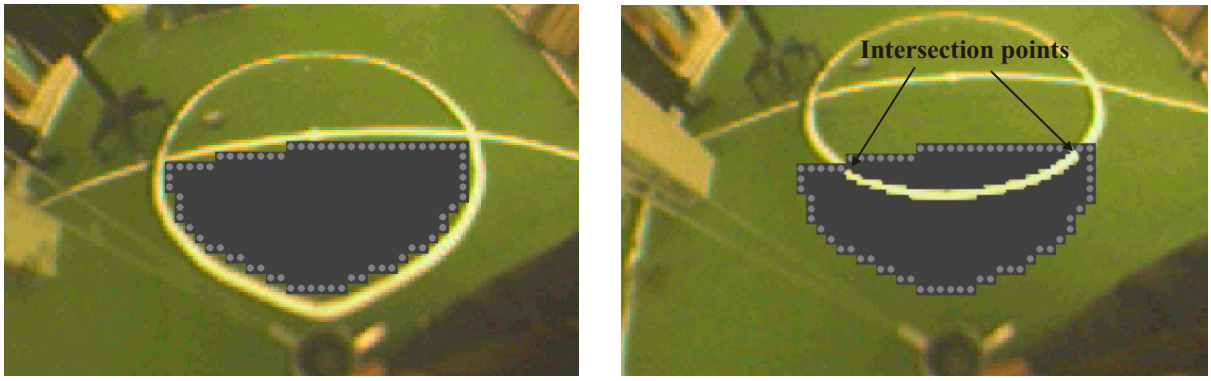


Figure 3.15: On the left a region has been grown and the resulting drops are shown. On the right, the robot has moved. The new image and the old region represent the initial state before shrinking. But none of the drops, except the ones at the intersection of the line and the region, will shrink, since they all lie on the playing field.

points of the old and new contours. These intersection points are labelled in figure 3.15b). Starting at these locations, the drops spread along the intruded line during the following shrinking phase as depicted in figure 3.16. In our RoboCup application, intersections of

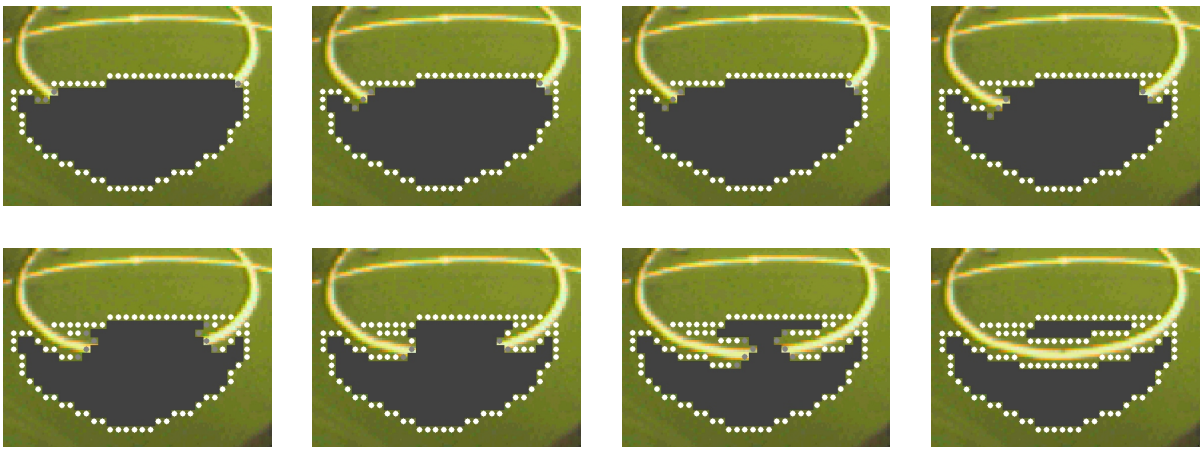


Figure 3.16: During the shrinking phase the drops uncover the line.

the field lines will always occur, since all field lines are connected with each other and their extension is greater than any region. Thus, not all the lines can intrude a region without cutting its boundary.

3.4 Results

The advantage of the proposed region tracking algorithm is that only the non-overlapping parts of corresponding regions are processed in successive images. However, if the regions don't overlap, the algorithm has to launch initial searches in each frame and degenerates. Therefore the question is, how often do the regions overlap? Of course, the answer depends on the application. In the following, we present results from our RoboCup application. Here, we have tracked three different types of regions: the green playing field and the blue and yellow regions (goal and post markings). While the robot moved through the environment, we computed the fraction of the processed area with respect to the size of the image and the percentage of overlap of the tracked regions. We determined these values for each pair of successive frames and built the respective average over a longer time span (see table 3.1). We also counted the number of initial searches and repeated the experiment for different movements (rotation/translation) and speeds of the robot.

Rotation	processed fraction of the image	% overlapping	frames	initial searches
$56°/s$	8.0%	80%	181	11
$74°/s$	8.4%	77%	142	11
$110°/s$	8.8%	71%	97	13
$145°/s$	10.0%	65%	74	13
$200°/s$	11.0%	60%	62	25
Translation	processed fraction of the image	% overlapping	frames	initial searches
$0m/s$	6.3%	93%	138	0
$0.26m/s$	5.9%	89%	114	0
$0.38m/s$	6.8%	84%	82	2
$0.52m/s$	7.0%	82%	61	0
$0.65m/s$	7.2%	80%	48	0
$0.74m/s$	6.9%	81%	42	1
$0.83m/s$	7.9%	75%	40	0
$1.0m/s$	7.8%	73%	31	1
$1.0m/s$	7.7%	79%	24	2

Table 3.1: The average fraction of the processed area with respect to the size of the image and the percentage of the overlapping area with respect to the area of the tracked regions has been determined for different movements and speeds. The last two columns indicate the number of frames and the number of initial searches that had to be started.

The tracked regions overlap significantly and only a small fraction of the image data

is accessed (below 10%). As expected, the rotation is more disadvantaged than the translation. This is because objects that are far away from the robot, such as the goals, have a high speed in the image, if the robot rotates. Therefore, the object regions might not overlap, and initial searches need to be executed. With a rotational speed of 200° per second, 25 initials searches had to be done within 82 frames. Assuming that an initial search requires accessing all the pixels in the image, the algorithm still performs well compared with common methods, which access all the pixels in each frame.

Although the theoretical running time of the proposed tracking algorithm is evidently faster than any algorithm that touches all the pixels in each image, the question is whether this is also true for the practical running time, since the maintenance of the drops, queues and connectivity grid might be more time consuming than a straightforward implementation.

Therefore, we decided to compare the algorithm with the color segmentation algorithm proposed in [15], which is believed to be the fastest color tracking algorithm at the moment, and which is applied by most teams in RoboCup. The algorithm is based on classifying each pixel into one or more of up to 32 color classes using logical *AND* operations and it employs a tree-based *union find* with path compression to group runs of pixels having the same color class.

We calibrated both algorithms to track green, blue and yellow regions. We did not track the orange ball and black obstacles, because our system uses different methods than color class tracking to recognize them. The following table gives the absolute running times of the algorithms over 1000 frames with the robot moving. Each image consist of 640×480 pixels (*YUV 4:2:2*) and we applied a Pentium III 800 Mhz processor.

CMU Algorithm	**Our Algorithm**
37.47 s	22.67 s

Thus, the proposed algorithm is significantly faster. Moreover, the proposed algorithm also extracts the contour curves of all regions at each frame.
However, it is not claimed that the algorithm performs better in all possible cases. There might be applications where CMU's algorithm performs better. This will typically be in cases where many small and fast moving regions are to be tracked.