
Blob Analysis

Mr Herbert was cleaning a cupboard when he found the blob of glue. His girlfriend noticed that it looked similar to Homer Simpson, and he decided to try to sell it on eBay.

[HTTP://WEB.ORANGE.CO.UK](http://web.orange.co.uk)

2.1 Introduction

In the previous chapter we have been looking into methods that allow us to extract pixel-precise regions corresponding to the objects present in the image. The obtained regions can be and usually are subject to inspection - measurements, classification, counting, etc. Such analysis of pixel-precise shapes extracted from image is called **Blob Analysis**, Region Analysis or Binary Shape Analysis.

Blob Analysis is a fundamental technique of image inspection; its main advantages include high flexibility and excellent performance. Its applicability is however limited to tasks in which we are able to reliably extract the object regions (see **Template Matching** for an alternative). Another drawback of the technique is pixel-precision of the computation (see **Contour Analysis** for a subpixel-precise alternative).

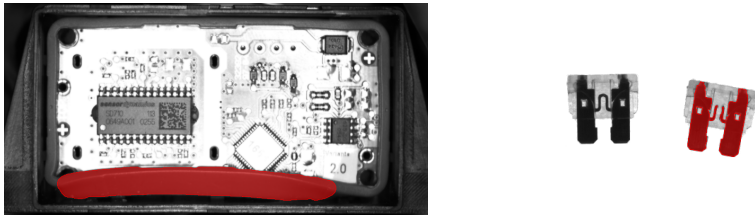


Figure 2.1: Example Blob Analysis applications - detection of excessive rubber band segment and disconnected fuses.

A typical Blob Analysis-based solution consists of the following steps:

1. **Extraction** - firstly a region corresponding to image objects is extracted from the image, usually by means of Image Thresholding.
2. **Processing** - secondly the region is subject to various transformations that aim at enhancing the region correspondence to actual object or highlighting the features that we want to inspect. In this phase the region is often split into connected components so that each one can be analyzed individually.

3. **Feature Extraction** - in the final part we may want to compute numerical and geometrical features describing the refined regions, such as its diameter, perimeter, compactness, etc. Such features may be the desired result themselves, or be used as a discriminant for region classification.

As Image Thresholding has already been discussed in the previous chapter, this chapter will focus entirely on two latter steps. We will commence with a demonstration of the data structure that we will use for region representation and then introduce a powerful set of transformation techniques called Mathematical Morphology.

After that we will review the numerical and geometrical features of binary shapes that are particularly useful for visual inspection. We will conclude the chapter with a handful of examples of the applications of Blob Analysis.

2.2 Region

Let us begin by defining the fundamental data type of pixel-precise binary shape, here called a region.

Region is **any** subset of image pixels.

This definition means that a region may represent any pixel-precise shape present in the image, connected or not, including empty region and full region. Image Thresholding operations discussed in the previous chapter return a single region - possibly representing a number of image objects.

Data Representation

The actual representation of a region in computer memory does not affect the theory of **Blob Analysis** but has important practical implications. Typically decision on the data representation boils down to the trade-off between memory efficiency of the data storage and computational efficiency of the operations that we intend to perform on data instances.

Binary Image

One trivial representation of a region would be a binary image, each of its pixels having a value of 0 (not-in-region) or 1 (in-region). Such representation

is quite verbose, as each region (even empty region) consumes an amount of memory corresponding to the size of the original image. On the other hand, this representation allows $O(1)$ lookup time for determining whether a pixel belongs to a given region.

Run-Length List

We could reduce the memory consumption using a classic data compression technique: Run-Length Encoding. In this technique consecutive, uniform sections (runs) of data are stored as tuples (*value, length*). We could adapt this idea to store binary images.

We will consider each row of the pixels separately and in each such row we will identify the runs of *one*-pixels (pixels belonging to the region). Each such run will be represented as a tuple ($x, y, length$). x and y will denote the coordinates of the first pixel of the run. As we are representing only the foreground pixels, we can discard the *value* component.

Such representation does not allow for $O(1)$ random-pixel access anymore, but as long as the list of pixel runs is sorted, we can achieve $O(\log(R))$ pixel lookup time, R denoting the number of pixel runs. In return this representation allows to perform various operations (such as region intersection or moment computation) in time dependent on the number of runs rather than number of pixels; which yields significant speed-up for typical regions.

As to memory efficiency, the results of a simple benchmark are presented in **Table 2.1**. We took into account three representations, each applied to store four regions extracted from 250x200 images.

- **Binary Image (uint8)** - a variant of Binary Image representation in which each pixel is stored as 0 or 1 value of 8-byte integer value. Although suboptimal, 8-byte per pixel is prevalent pixel depth for such applications because of the low-level details of memory access.
- **Binary Image (bit)** - a variant of Binary Image representation in which each pixel is stored as 0 or 1 value of a single bit.
- **Run-Length Encoding** - we assumed that each element of the ($x, y, value$) tuple is stored using 16-bit integer type, which accumulates to 6-bytes per pixel run memory usage.

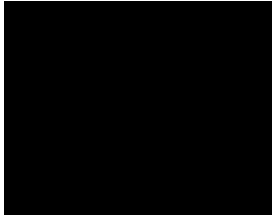

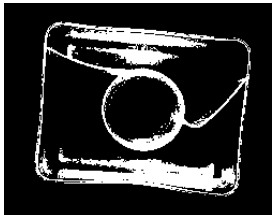
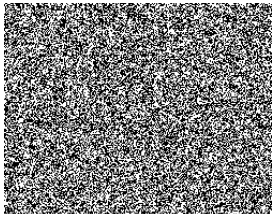
	Image (uint8)	Image (bit)	RLE
	50000	6250	0
	50000	6250	978
	50000	6250	6276
	50000	6250	75102

Table 2.1: Number of bytes consumed by different region representations.

Region Dimensions

In our reference implementation, **Adaptive Vision Studio 2.5**, regions are represented using Run-Length Encoding described above, with one slight extension: each region stores not only the set of pixels in form of list of runs, but also two integers representing its reference dimensions: width and height.

These are usually the dimensions of image the region was extracted from and serve two purposes. For one thing, they allow meaningful display of a region in the context of image it refers to; for other thing - they conveniently allow to define a complement of a region.

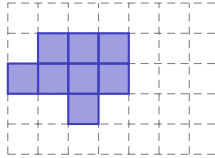


Figure 2.2: Region of dimensions: 7 (width), 5 (height)

2.3 Basic Operations

In this section we will introduce six elementary operations that can be performed on a region. Four of them refer to set nature of a region (being essentially a set of pixels), two further are defined in relation to its spatial properties.

In the next section we will use these building blocks to define a set of way more powerful transformations from the field of Mathematical Morphology.

Set Operators

Applicability of basic set operators to region processing follows directly from the definition of region.

Union

Union of two regions is a region containing the pixels belonging to either, of both of the input regions, as demonstrated in **Table 2.2**.

Intersection

Similarly, intersection of two regions is a region containing the pixels belonging to both of the input regions, as demonstrated in **Table 2.3**.

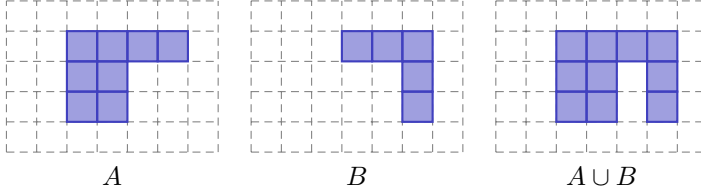


Table 2.2: Union of two regions

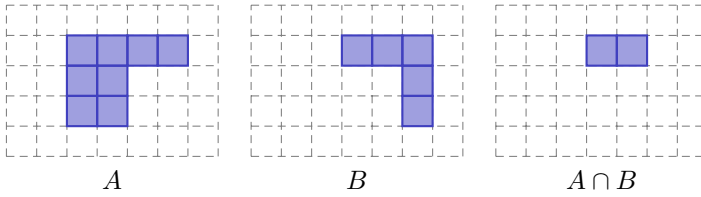


Table 2.3: Intersection of two regions

Difference

Last binary operation in this set is difference, yielding the pixels belonging to first region, but not to the second region. Thus, this operation is not commutative, contrary to intersection and union.

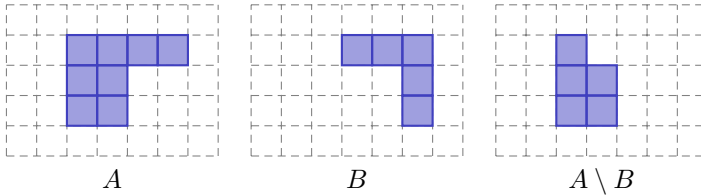


Table 2.4: Difference of two regions

Complement

The only unary set operator, complement, is also applicable to region; however industrial implementations differ in its interpretation. We will follow the

way of **Adaptive Vision Studio 2.5**, where complement is easy to define as each region stores the dimensions of its finite reference space.

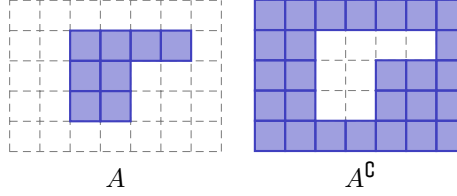


Table 2.5: Complement of a region

Spatial Operators

Two further operators refer to spatial properties of region. Naturally, there are far more spatial operators than can be defined for region; for now we introduce only two that are necessary to define morphological operators discussed in the next section.

Translation

Translation of a region shifts its pixel coordinates by integer vector.

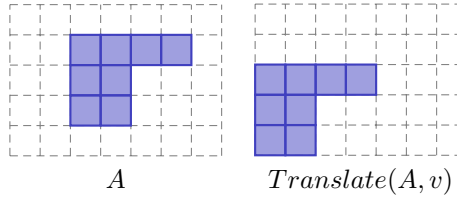


Table 2.6: Translation of a region by vector $-2, 1$.

Reflection

Reflection mirrors a region over a location (origin), marked with a black square in **Table 2.7**. This operation will be particularly useful for processing morphological kernels, which we will discuss in the next section.

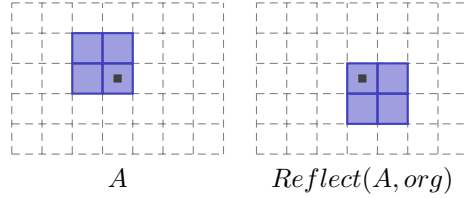


Table 2.7: Reflection of a region.

2.4 Mathematical Morphology

Mathematical Morphology, born in 1960s and rapidly developing ever since, is both a theory and technique for processing spatial structures. Soille described[6] the field as being *mathematical* in that it is built upon set theory and geometry, and being *morphology*¹ in that it aims at analyzing the shape of objects.

In most general case, Mathematical Morphology applies to any complete lattice. We will concentrate on its application to region processing. In this context Mathematical Morphology can be looked at as a set of techniques that alter the region by probing it with another shape called kernel or structuring element.

Kernel

Kernel in Mathematical Morphology is a shape that is repeatedly aligned at each position within the dimensions of the region being processed. At each such alignment the operator verifies how the aligned kernel fits in the region (e.g. if kernel is contained in the region in its entirety) and depending on the results includes the position in the results or not.

As kernel is pixel-precise binary shape itself, it can be represented as a region together with integer coordinates of its origin. Specifying the origin is important, as it is the position that will be aligned against the region being processed.

¹From Greek *morphe* meaning form.

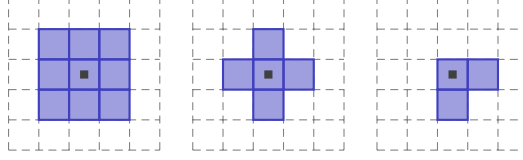


Table 2.8: Example kernels for morphological operations.

Dilation

First morphological operation that we are going to discuss is dilation. In this operator the kernel aligned at each position within the region dimensions needs to overlap with at least one pixel of the input region to include this position in the result:

$$Dilate(R, K) = \{[p_x, p_y] | R \cap Translate(K, [p_x, p_y]) \neq \emptyset\}$$

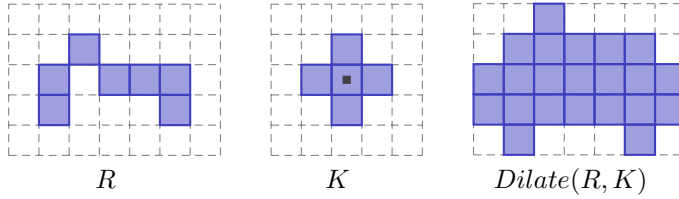


Table 2.9: Dilation of a region

If we decompose the kernel into its individual pixels we may observe that each such pixel $[k_x, k_y] \in K$ contributes a copy of the region translated by $[-k_x, -k_y]$ into the result. Therefore we may also define the dilation operator as follows:

$$Dilate(R, K) = \bigcup_{[k_x, k_y] \in K} Translate(R, [-k_x, -k_y])$$

Dilation effectively expands the region, the magnitude and direction of the expansion depending on the kernel being used. The operator is commonly used to join disconnected components of a region. Dilating a region by circular kernel

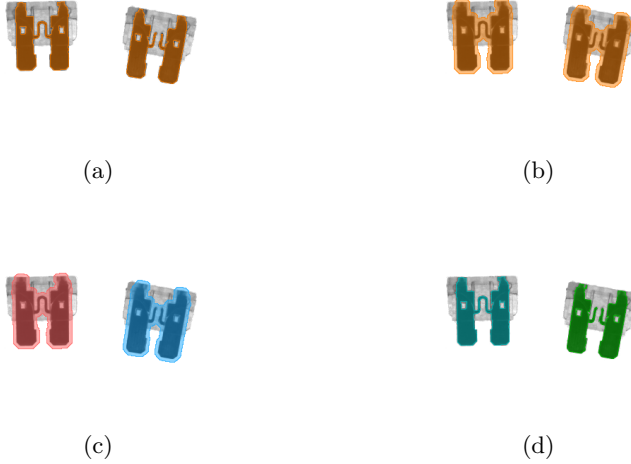


Figure 2.3: Dilation, extraction of connected components and intersection applied to split a region representing metal parts of the fuses (a) into components representing individual fuses (d).

of radius r will expand the region uniformly in each direction up to distance of r pixels, effectively joining region components separated by less than $2r$ pixels. One possible application is demonstrated in **Figure 2.3**.

In this example we process a region representing metal parts of two fuses. As one of the fuses is burned out, the region contains three connected components. To split it into two connected components, each representing an individual fuse, we may perform the dilation before extracting the region components and intersect the resulting regions with the original one to preserve their original shape.

Erosion

Erosion is shrinking counterpart of dilation. This operator requires that the aligned kernel is fully contained in the region being processed:

$$Erode(R, K) = \{[p_x, p_y] | Translate(K, [p_x, p_y]) \subseteq R\}$$

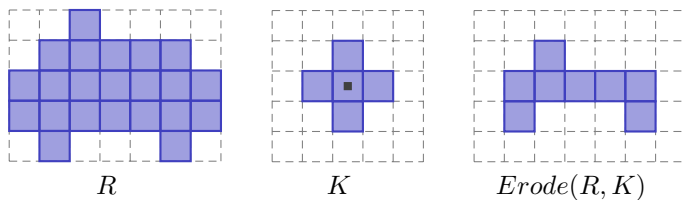


Table 2.10: Erosion of a region

Similarly to dilation, we may also formulate erosion in terms of kernel decomposition. In this case each pixel of the kernel $[k_x, k_y] \in K$ also contributes the shifted copy of a region, but a position must be contained in all such contributions to be included in the results:

$$Erode(R, K) = \bigcap_{[k_x, k_y] \in K} Translate(R, [-k_x, -k_y])$$

The operations of dilation and erosion are closely related, but it is important to note that they are not inverse² of each other, i.e., erosion of a region does not necessarily *cancel out* previously applied dilation; counterexample being presented in **Table 2.9** and **Table 2.10**. Quite contrary, consecutive application of dilation and erosion is extremely useful operation and will be discussed soon.

Although dilation is not an inverse of erosion, another relation between the operations holds - they are duals of each other, meaning that dilation of a region is equivalent to erosion of its background (complement), and conversely.

$$Erode(R, K) = Dilate(R^c, K)^c$$

²Actually neither of these operation has an inverse, as such operation would have to magically guess where the lone pixels lost in erosion or holes filled in dilation were located.

Closing

Before we define the next operator, let us get back for a moment to the dilation operator. As we remember, dilation expands the region in the way defined by the structuring element. It is worth noting that during this expansion small holes and region cavities may get completely filled in. This effect is worth attention as filling gaps of a region³ is a common need in industrial inspection.

Unfortunately, dilation does not address this need precisely - the missing parts gets filled in, but also the region boundaries are expanded. It would be more convenient to have an operator that avoids the second effect while keeping the first.

The closing operator addresses this need by dilating the region and eroding it right after that:

$$Close(R, K) = Erode(Dilate(R, K), Reflect(K))$$

Initial dilation fills in the region gaps and the succeeding erosion brings the expanded region back to its original dimensions (but does not restore the gaps that were completely filled in).

It is worth noting that we use the reflected kernel for the second operation - if we recall that dilation may be formulated as a union of translations corresponding to individual pixels of the kernel ($\bigcup_{[k_x, k_y] \in K} Translate(R, [-k_x, -k_y])$), it is clear that we need to use the opposite translations to keep the region in its position.

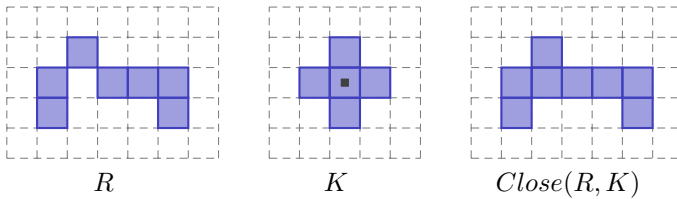


Table 2.11: Closing of a region

³Which could be introduced for instance by local glare of the lightning affecting the results of thresholding.

Closing is commonly applied whenever the extracted region contains gaps or cavities that should be filled in, an example of such application is demonstrated in **Figure 2.4**.

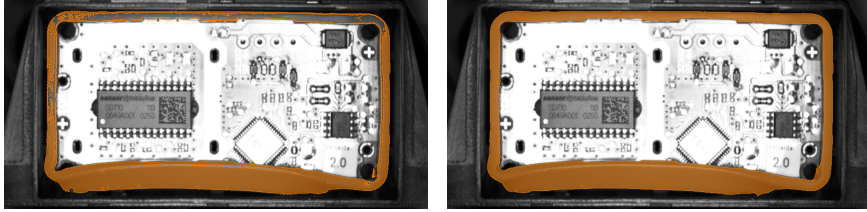


Figure 2.4: Closing operator used to fill gaps in a region.

Opening

Another useful morphological operator is obtained by interchanging the order of operators that closing is composed of. The opening operator firstly erode a region and then dilates the result:

$$Open(R, K) = Dilate(Erode(R, K), Reflect(K))$$

The effect of such composition is dual to the closing operator that we recently discussed. The initial erosion shrinks the region removing its isolated pixels and small branches, while the successive dilation brings it back to original dimensions, but cannot restore the parts that vanished completely during erosion.

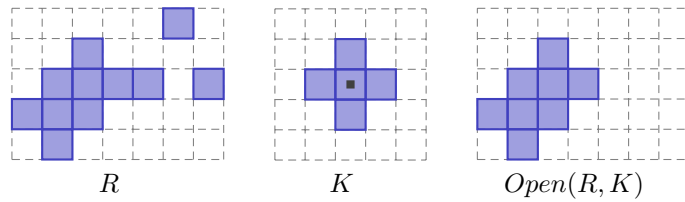


Table 2.12: Opening of a region

The opening operator may be applied to remove salt noise in the region or to eliminate its thin parts. Opening a region using a circular kernel of radius r will remove all segments of the region that have less than $2r$ pixels in width (and keep the other parts intact). An example application is demonstrated in **Figure 2.5**.

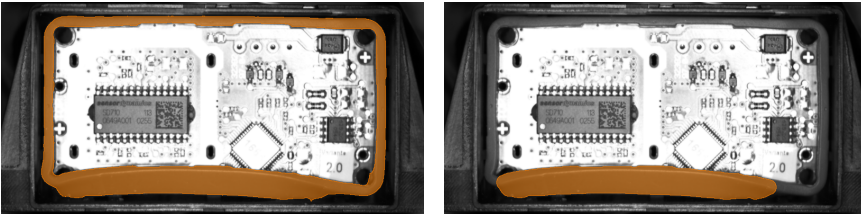


Figure 2.5: Opening operator used to determine excessively wide section of the rubber band.

2.5 Features

Numerical

Various shape properties may be expressed in form of numeric descriptors. For instance, if we divide the area of a region by the area of its bounding rectangle, we get its rectangularity factor - a real number between 0.0 and 1.0 reflecting how similar a region is to rectangle.

Table 2.13 contains a list of particularly useful region features.

Feature	Range	Description
Area	$(0 - \infty)$	Number of pixels in the region.
Circularity	$(0.0 - 1.0)$	Similarity to a circle computed as the region area divided by area of circular region with the same radius.
Compactness	$(0.0 - 1.0)$	Similarity to a circle computed as the region area divided by area of circular region with the same perimeter length.
Convexity	$(0.0 - 1.0)$	Convexity factor computed as the region area divided by area of its convex hull.
Elongation	$(0.0 - \infty)$	Relation of region length to its width.
Moments	$(-\infty - \infty)$	(..)
Number of holes	$(0 - \infty)$	(..)
Orientation	$(0.0 - 180)$	Orientation of region main axis of inertia.
Perimeter length	$(0.0 - \infty)$	Length of the region boundary.
Rectangularity	$(0.0 - 1.0)$	Similarity to a rectangle computed as the region area divided by area of its bounding rectangle.

Table 2.13: Numeric properties of region shape.