Chapter 5

THE DETECTION OF CONNECTED COMPONENTS

§5.1 Preliminaries

In the preceding two chapters, we have studied the 1- and 2-dimensional building blocks or structural elements of a figure, namely *runs* and *objects* together with their adjacency relations. We also showed that exhaustive information about these structural elements can be acquired in real-time.

In this chapter, we turn to the consideration of the third level in our hierarchical decomposition of figures, namely, the *connected components* and their adjacency relations. The real-time requirement can presently be formulated as follows: A connected component of the figure must be detected at the end of the scan of the last run contained in it, and the amount of processing involved must be proportional to the sum of the sizes of the records associated with the objects contained in that component.

Other approaches have used the structure of the cycles formed by the *border* of the figure to ascertain that connected components or holes are completely

disclosed [2,3]. Indeed, a connected component is completely disclosed when its outer border, which forms a cycle, is closed. In very much the same way, we shall use the closure of the *edge* for that purpose. To this end, we shall keep track of *chains*, *i.e.*, unclosed cycles, in the edge, together with the pairs of their current endpoints. Clearly, coincidence of endpoints will reveal the closure of the edge cycles.

Thus far, the description of our approach has a definite "general purpose" flavor. This point is well exemplified by the redundancy in the information stored in the object records. It is quite clear, however, that one should not use the same program for the purposes of isolating characters in typed text and analyzing the topology of conducting tracks in sophisticated printed circuit boards. It is at the present stage that we shall encounter two possibilities of specializing our technique to the needs and taste of a wide range of potential users. Both these specializations offer two ways of acquiring and handling information about *adjacency* relations and *surrounding* relations respectively. It should be stressed at the outset that they are fully orthogonal in the sense that any combination is feasible (and directly available in the code).

For what concerns adjacency, we introduce two options called *full adjacency*, and *restricted adjacency*. With the former, we use exclusively—and extensively—the ten object parameters $pre(suc)cnnb$, $prefi(la)$, $preletori(ritole)$, $sucfi(la)$, $sucletori(ritole)$ for the purposes of following the edges of connected components and holes, and detecting the closure of edge cycles. Thus, the *full adjacency* option is the natural continuation of the approach discussed up to now. However, we made it clear that these parameters offer us more than what we actually need. Thus, with the *restricted adjacency* option we eliminate the redundancy in the contents of object records, and we show that edges can be followed and closures detected by using a different approach which completely ignores these ten parameters but uses only four new parameters defined in Subsection 5.2.3. As the name implies, *full adjacency* is intrinsically richer and facilitates subsequent analysis of connected components, for instance, the recognition stage in an OCR application. On the other hand *restricted adjacency* yields a less detailed output but results in a simplified program.

The second specialization concerns surrounding relations. Here again two options are available, namely *full surrounding* and *restricted surrounding*. Latter focuses on connected components taken in isolation. In other words, under *restricted surrounding*, we extract only the surrounding relations between every connected component and its respective holes. On the contrary, under *full surrounding*, we extract the surrounding relations between all connected components of

the figure and all connected components of the background except the one which contains the grid (as it is plain that this one surrounds everything anyway).

This chapter discusses a variety of subjects which will be brought to bear on the design of the program in a number of different ways. Therefore, a brief overview should prove helpful.

Section 5.2 is concerned with edge-following in a figure. Brief theoretical developments are followed by minute examination of *full* and *restricted adjacencies*. Accordingly, we describe two versions of an edge-following operator which will be called into action at the time we want to output the cycles of the edge of the figure.

In Section 5.3, we introduce the *neighborhood-tree* as a representation for neighborhood and surrounding relations in a figure. In turn, neighborhood-trees can be represented in the form of *strings*. We discuss two possible string representations, namely, the *vertex-string* and the *edge-string*. We provide evidence that the latter proves more convenient for our purposes, and we discuss edge-strings in both *full* and *restricted surroundings*. Eventually, we introduce the data structure which is used to store cycles and edge-strings.

Detection of the closure of cycles and construction of edge-strings is the subject matter of Section 5.4. We show how cycles can be found by growing *chains* (parts of cycles) in the edge of the figure. We introduce the data structure required to store chain extremities together with their properties under both *full* and *restricted surroundings*, and we show that the operations to be performed on chains and edge-strings can be conveniently distributed between six new subroutines which are again nested in procedure *allocate*.

§5.2 Edge-Following in a Figure

5.2.1 Notation and theoretical background

We first introduce the notation for the *edge* of a figure, and we recall some basic properties which will be useful in the sequel. It should be borne in mind that we constantly assume that F satisfies the frame assumption (see Section 2.2).

Figure 5.1. Oriented (x, y), and unoriented $\{x, y\}$ edge-elements.

Let X and Y be two neighboring subsets of the grid G. Then, the *oriented edge* $\epsilon^+(X, Y)$ between X and Y is the set of all ordered pairs (x, y) such that $x \in X$, $y \in Y$ and x is 4-adjacent to y. Such an ordered pair (x, y) is called an (oriented) *edge element*, and is represented in Figure 5.1.a as an arrow between x and y. The orientation of the edge $\epsilon^+(X, Y)$ is such that the set of arrows leaves X on the left and Y on the right.

One can also define the *unoriented edge* $\epsilon(X, Y)$ between X and Y. It consists of all unordered pairs $\{x, y\}$, and such a pair is represented in Figure 5.1.b as a straight bar between x and y. Obviously, we could also use the inverse orientation of the edge, leaving X on its right and Y on its left, and denoted $\epsilon^-(X, Y)$. In fact, we have

$$\epsilon^-(X, Y) = \epsilon^+(Y, X), \tag{5.1}$$

and

$$\epsilon(Y, X) = \epsilon(X, Y). \tag{5.2}$$

We define

$$\epsilon^+(X) \doteq \epsilon^+(X, G\backslash X), \tag{5.3}$$

and similarly for $\epsilon^-(X)$ and $\epsilon(X)$. These are the positive oriented, unoriented and negative oriented edges of X. Note that

$$\epsilon^-(X) = \epsilon^+(G\backslash X). \tag{5.4}$$

Let us now consider the edge $\epsilon^+(F)$ of figure F, or equivalently the edge $\epsilon^-(B)$ of the background $B = G\backslash F$. Let X denote a k-connected component of

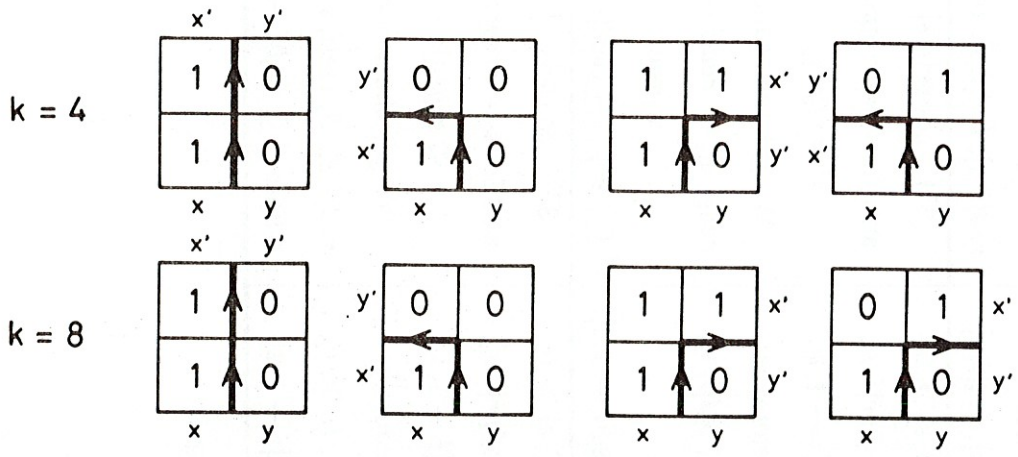


Figure 5.2. The successor of an edge-element.

F and Y denote a k'-connected component of B. Now, $\epsilon^+(F)$ is the disjoint union of $\epsilon^+(X, Y)$ for all such X's and Y's with Y neighboring X. It is also the union of $\epsilon^+(X)$ for all such X's, and the union of $\epsilon^-(Y)$ for all such Y's. Furthermore, $\epsilon^+(F)$ is a union of *cycles*, and conversely, these cycles are the $\epsilon^+(X, Y)$ obtained for all possible pairs (X, Y).

The cycle $\epsilon^+(X, Y)$ can be obtained from one of its edge-elements (x_0, y_0) by successive application of the *edge-following operator*, which is a permutation of $\epsilon^+(X, Y)$ associating to each edge-element another edge-element which is its successor in the cycle. Thus, starting from (x_0, y_0), we obtain successively (x_1, y_1), (x_2, y_2), $\ldots$, $(x_n, y_n) = (x_0, y_0)$ and the cycle is completed.

Figure 5.2 displays the successor (x', y') of (x, y) for all possible configurations over a 2×2-window containing (x, y), and for both $k = 4$ and $k = 8$.

In the next two subsections we shall address the problem of devising real-time, edge-following operators for the objects we are dealing with. In accordance with our discussion in Section 5.1. *full* and *restricted adjacencies* will be examined separately.

A. Following $led[i,s]$	B. Preceding $red[i,s]$	C. Preceding $led[i,s]$	D. Following $red[i,s]$
(1°) $consu[i,s]=0$ We get the sequence $led[i,s]$, $bed[i,s]$, $red[i,s]$	(1°) $consu[i,s]=0$ We get the sequence $led[i,s]$, $bed[i,s]$, $red[i,s]$	(1°) $conpr[i,s]=0$ We get the sequence $red[i,s]$, $ted[i,s]$, $led[i,s]$	(1°) $conpr[i,s]=0$ We get the sequence $red[i,s]$, $ted[i,s]$, $led[i,s]$
(2°) $consu[i,s]>0$ and for $t := lefsu[i,s]$, $s = lefpr[i+1,t]$. We get the sequence $led[i,s]$, ϵ, $led[i+1,\tau]$, where $\epsilon \subseteq bed[i,s] \bigcup ted[i+1,t]$.	(2°) $consu[i,s]>0$ and for $t := rigsu[i,s]$, $s = rigpr[i+1,t]$. We get the sequence $red[i+1,t]$, ϵ, $red[i,s]$, where $\epsilon \subseteq bed[i,s] \bigcup ted[i+1,t]$.	(2°) $conpr[i,s]>0$ and for $t := lefpr[i,s]$, $s = lefsu[i-1,t]$. We get the sequence $led[i-1,t]$, ϵ', $led[i,s]$, where $\epsilon' \subseteq ted[i,s] \bigcup bed[i-1,t]$.	(2°) $conpr[i,s]>0$ and for $t := rigpr[i,s]$, $s = rigsu[i-1,t]$. We get the sequence $red[i,s]$, ϵ', $red[i-1,t]$, where $\epsilon' \subseteq ted[i,s] \bigcup bed[i-1,t]$.
(3°) $consu[i,s]>0$ and for $t := lefsu[i,s]$, $s > lefpr[i+1,t]$. We get the sequence $led[i,s]$, τ, $red[i,s-1]$, where $\tau \subseteq ted[i+1,t]$.	(3°) $consu[i,s]>0$ and for $t := rigsu[i,s]$, $s < rigpr[i+1,t]$. We get the sequence $led[i,s+1]$, τ, $red[i,s]$, where $\tau \subseteq ted[i+1,t]$.	(3°) $conpr[i,s]>0$ and for $t := lefpr[i,s]$, $s > lefsu[i-1,t]$. We get the sequence $red[i,s-1]$, β, $led[i,s]$, where $\beta \subseteq bed[i-1,t]$.	(3°) $conpr[i,s]>0$ and for $t := rigpr[i,s]$, $s < rigsu[i-1,t]$. We get the sequence $red[i,s]$, β, $led[i,s+1]$, where $\beta \subseteq bed[i-1,t]$.

Table 5.1. Edge-following at the run-level

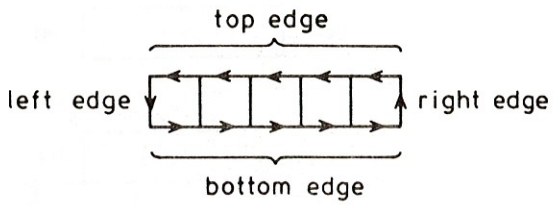


Figure 5.3. The edge of a run.

5.2.2 Full adjacency

Let us recall that under *full adjacency* all of the object parameters defined thus far are available and can therefore be brought to bear on the specification of our edge-following operator. To simplify matters, we shall first examine this problem at the run-level.

In Figure 5.3, we show how the edge $\epsilon^+(R)$ of a run R can be subdivided into 4 parts called the *top*, *left*, *bottom* and *right* edges of R. If $R = run[i, s]$, then these 4 parts will be written $ted[i, s]$, $led[i, s]$, $bed[i, s]$ and $red[i, s]$ respectively.

We will henceforth concentrate particularly on left and right edges. We do so because the left and right edges of a run R of F always belong to the edge of F, while this is not always true of the top and bottom edges of R (since R may have adjacent runs of F from above or below).

Next, let us consider $run[i, s]$ embedded in some connected component of F. We already know that $led[i, s]$ and $red[i, s]$ belong to some cycle(s) in $\epsilon^+(F)$. Let us now examine what follows or precedes them, up to the next $led[j, t]$ or $red[j, t]$ in this or these cycles. A close examination of all possible configurations of adjacent runs reveals that the answer to that question follows from the consideration of only three configurations. This exhaustive analysis gives rise to Table 5.1 which shows the parts of $\epsilon^+(F)$ which follow or precede $led[i, s]$ and $red[i, s]$ in each of the three configurations of interest. Columns A and D of Table 5.1 are illustrated by Figure 5.4. Columns B and C of that table specify what precedes the left or right edge of $run[i, s]$, and are merely illustrative.

Table 5.1 demonstrates that the sequence of left and right edges of the runs of F in $\epsilon^+(F)$ is completely determined by the parameters of these runs. However, it seems worthwhile to point out that, at the run-level, the information

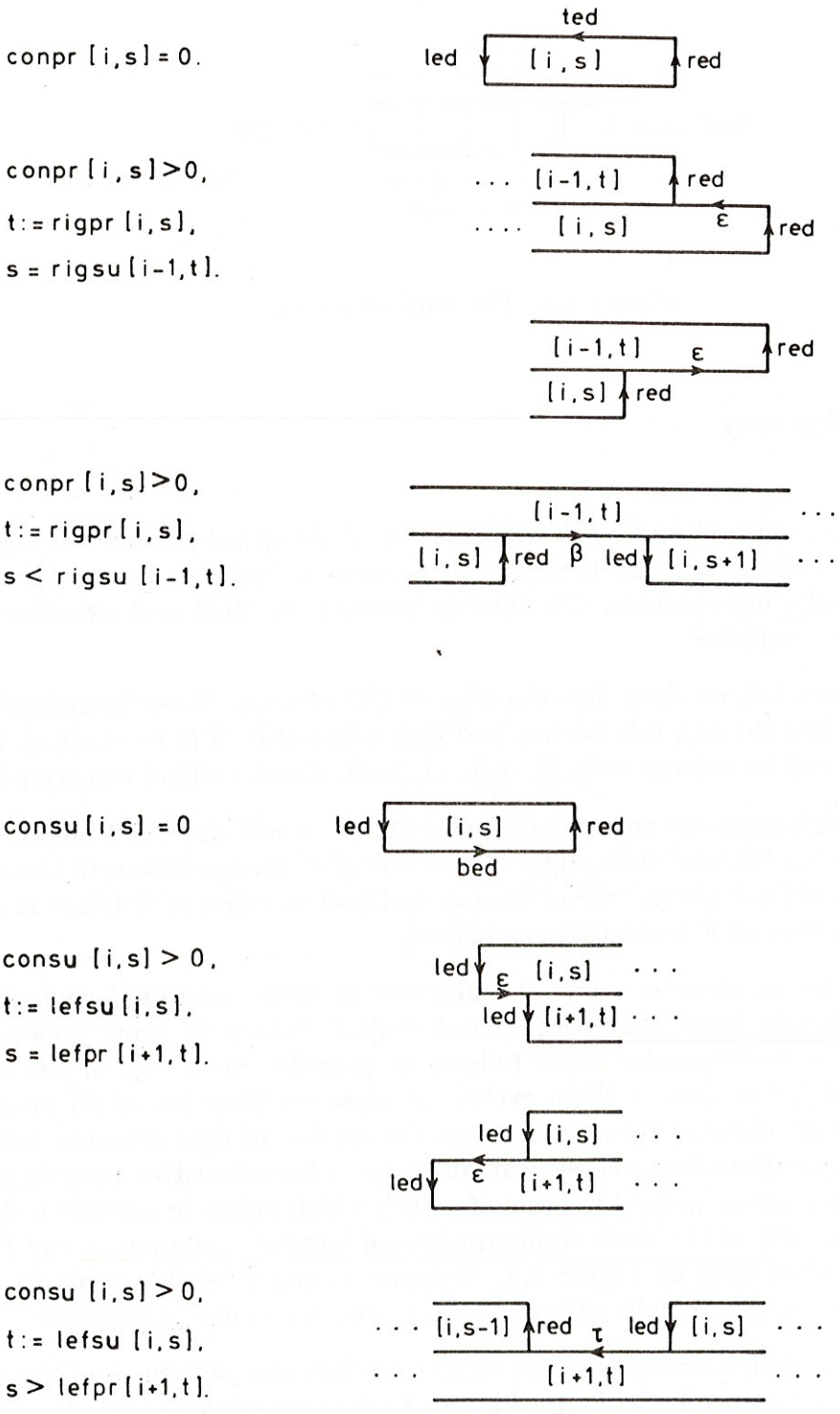


Figure 5.4. Edge-following at the run level

required to achieve this goal can sometimes be accessed only in an indirect manner. This point is well apparent in cases $A.3°$ and $D.3°$. There, the edge elements $red[i, s-1]$ and $led[i, s+1]$ which follow $led[i, s]$ and $red[i, s]$ can be found only by inspection of the parameters of $run[i-1, t]$ and $run[i+1, t]$ respectively. This observation provides most of the justification for the redundancy that was introduced in the object parameters. As we shall see hereafter, object parameters will enable us to follow the edge of F in a more straigthforward manner.

Let us now apply these properties of runs to our 2-dimensional objects: hinges, blocks and block-continuations. Consider first a block X formed by $1+w$ block runs: $run[i, t_0], \ldots, run[i, t_w]$. Then, the edge $\epsilon^+(X)$ of X consists in 4 parts:

- The top edge $ted[i, t_0]$.
- The bottom edge $bed[i, t_w]$.
- The sequence: $led[i, t_0], \epsilon_0, \ldots, \epsilon_{w-1}, led[i+w, t_w]$, where $\epsilon_0, \ldots, \epsilon_{w-1}$ are as ϵ in Table 5.1.A. (5.5)
- The sequence: $red[i+w, t_w], \epsilon'_w, \ldots, \epsilon'_0, red[i, t_0]$, where $\epsilon'_w, \ldots, \epsilon'_0$ are as ϵ' in Table 5.1.D. (5.6)

The sequences (5.5) and (5.6) are the *left* and *right* edges of block X. We illustrate in Figure 5.5 the edge of a block. Now, we can obviously proceed in the same manner for a block-continuation. On the other hand, a hinge is a run, and so what was said above about the edge of a run applies to hinges, except for the fact hinges are described by objects parameters instead of run parameters. Thus the edge of a 2-dimensional object can again be subdivided into 4 parts, the *top*, *left*, *bottom* and *right* parts. Given an object S, we will write them $ted(S)$, $led(S)$, $bed(S)$ and $red(S)$ respectively.

Adjacency relations between objects correspond to adjacency relations of their topmost and bottommost runs. This enables us to translate the (run-based) Table 5.1 in the (object-based) Table 5.2 which specifies how the edge of F can be followed at the object-level. Here again, columns A and D provide essential information with regard to the algorithm, while columns B and C are illustrative. Table 5.2 shows that the sequences of left and right edges of objects of F are completely determined by the adjacency relations between these objects, as they are expressed by the ten object parameters. Let us also note that these parameters

A. Following $led(S)$	B. Preceding $red(S)$	C. Preceding $led(S)$	D. Following $red(S)$
(1°) $succnnb[S] = 0$ We get the sequence $led(S)$, $bed(S)$, $red(S)$	(1°) $succnnb[S] = 0$ We get the sequence $led(S)$, $bed(S)$, $red(S)$	(1°) $precnnb[S] = 0$ We get the sequence $red(S)$, $ted(S)$, $led(S)$	(1°) $precnnb[S] = 0$ We get the sequence $red(S)$, $ted(S)$, $led(S)$
(2°) $succnnb[S] > 0$ and for $T := sucfi[S]$, $S = prefi[T]$. We get the sequence $led(S)$, ϵ, $led(T)$, where $\epsilon \subseteq bed(S)$ or $ted(T)$.	(2°) $succnnb[S] > 0$ and for $T := sucla[S]$, $S = prela[T]$. We get the sequence $red(T)$, ϵ, $red(S)$, where $\epsilon \subseteq bed(S)$ or $ted(T)$.	(2°) $precnnb[S] > 0$ and for $T := prefi[S]$, $S = sucfi[T]$. We get the sequence $led(T)$, ϵ', $led(S)$, where $\epsilon' \subseteq ted(S)$ or $bed(T)$.	(2°) $precnnb[S] > 0$ and for $T := prela[S]$, $S = sucla[T]$. We get the sequence $red(S)$, ϵ', $red(T)$, where $\epsilon' \subseteq ted(S)$ or $bed(T)$.
(3°) $succnnb[S] > 0$ and for $T := sucfi[S]$, $S \neq prefi[T]$. We get the sequence $led(S)$, τ, $red(S')$, where $\tau \subseteq ted(T)$, and $S' := preritole[S]$.	(3°) $succnnb[S] > 0$ and for $T := sucla[S]$, $S \neq prela[T]$. We get the sequence $led(S'')$, τ, $red(S)$, where $\tau \subseteq ted(T)$, and $S'' := preletori[S]$.	(3°) $precnnb[S] > 0$ and for $T := prefi[S]$, $S \neq sucfi[T]$. We get the sequence $red(S')$, β, $led(S)$, where $\beta \subseteq bed(T)$, and $S' := sucritole[S]$.	(3°) $precnnb[S] > 0$ and for $T := prela[S]$, $S \neq sucla[T]$. We get the sequence $red(S)$, β, $led(S'')$, where $\beta \subseteq bed(T)$, and $S'' := sucletori[S]$.

Table 5.2. Edge-following at the object-level

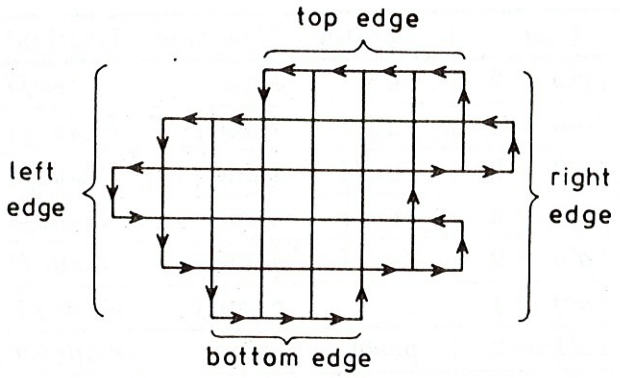


Figure 5.5. A block and its edge.

give us now direct access to the information required for that purpose. (The alert reader will soon notice that the information which was lacking at the run-level was that conveyed by the parameters *preritole*(*letori*) and *sucritole*(*letori*), one of which is defined $\neq \emptyset$ in each occurrence of case 3°.)

In the cycles of the edge $\epsilon^+(F)$, the three cases occuring in colums A and D of Table 5.2 can be determined from the parameters of the current topmost and bottommost runs of each object. Therefore, it should come as no surprise that it is possible to determine, within the procedure *allocate*, which case of (1°), (2°) or (3°) holds for the successor of the left and right edge of a given object. This will bring about a first, though natural expansion of this procedure. However, before getting immersed into the implementational details of Subsection 5.2.4., it will be convenient to examine how the succession of left and right edges of objects in the cycles of $\epsilon^+(F)$ can be followed with the help of this decomposition in 3 cases.

It has proved useful to write in the object records the case (1°), (2°) or (3°) which holds for the successor of the left and right edges of that object. This can be done by adding to these records two components called $fol0$ and $fol1$ taking values in the subrange 1. .3. Here, 0 stands for "left", and 1 for "right". Thus $fol0$ indicates the number of the case which holds for the successor of the left edge of the object (in Table 5.2.A), while $fol1$ indicates the number of the case for the successor of the right edge of the object (in Table 5.2.D). Addition of these two components eliminates the need to recompute the cases when one wants to follow the cycles of $\epsilon^+(F)$.

Record	Edge side	Case	Pointer	New side	Destination
$r1$	left	$fol0 = 2$	$sucfi$	same	$r1.sucfi \uparrow = r2$
$r2$	left	$fol0 = 2$	$sucfi$	same	$r2.sucfi \uparrow = r3$
$r3$	left	$fol0 = 2$	$sucfi$	same	$r3.sucfi \uparrow = r5$
$r5$	left	$fol0 = 2$	$sucfi$	same	$r5.sucfi \uparrow = r6$
$r6$	left	$fol0 = 2$	$sucfi$	same	$r6.sucfi \uparrow = r7$
$r7$	left	$fol0 = 1$	...	change	$r7 = r7$
$r7$	right	$fol1 = 2$	$prela$	same	$r7.prela \uparrow = r6$
$r6$	right	$fol1 = 2$	$prela$	same	$r6.prela \uparrow = r5$
$r5$	right	$fol1 = 2$	$prela$	same	$r5.prela \uparrow = r4$
$r4$	right	$fol1 = 2$	$prela$	same	$r4.prela \uparrow = r2$
$r2$	right	$fol1 = 2$	$prela$	same	$r2.prela \uparrow = r1$
$r1$	right	$fol1 = 1$	...	change	$r1 = r1$

Table 5.3. Following the outer cycle in Figure 4.4.

Record	Edge side	Case	Pointer	New side	Destination
$r4$	left	$fol0 = 3$	$preritole$	change	$r4.preritole \uparrow = r3$
$r3$	right	$fol1 = 3$	$sucletori$	change	$r3.sucletori \uparrow = r4$

Table 5.4. Following the inner cycle in Figure 4.4

Let us show on an example how this allows us to follow the cycles of the edge of F. Consider the decomposition of Figure 4.4 into 7 objects numbered from 1 to 7, and whose records are labelled $r1, \ldots, r7$ respectively. The edge of the figure has two cycles. They can be followed with the help of a binary variable *side* representing the edge side of the object. Tables 5.3 and 5.4 show how this can be done.

With Table 5.2 handy, this example is fairly self-explanatory. Hence the details are left to the reader. One sees that the variable *side* determines whether one examines $fol0$ (left) or $fol1$ (right). In fact, Table 5.2 was organized in such a way that this variable changes its value when $fol0$ or $fol1$ is odd. This example also shows that, aside from the determination of the case, edge-following can be

implemented with the help of only four out of the ten object parameters. We shall pursue this idea further in the next subsection.

5.2.3 Restricted adjacency

As mentioned before, the *restricted adjacency* option completely ignores the ten objects parameters but uses instead four other parameters which we shall presently introduce. Thus, our goal becomes that of collecting in the object records the minimum information necessary to follow the succession, in $\epsilon^+(F)$, of left and right edges of objects. For this purpose, we must only specify which edge (left or right) of which object follows the left or the right edge of a given object. This can be done by introducing in the object records the following four components

$$\begin{aligned} &fol0poin,\ fol1poin : link, \\ &fol0side,\ fol1side : binary, \end{aligned} \tag{5.7}$$

where $binary = 0..1$ and $link = \uparrow objrec$.

With these four components we proceed as follows: The left edge of an object r is followed by the left or right edge of the object $r.fol0poin\uparrow$ according to whether $r.fol0side = 0$ or 1. Likewise, the right edge of r is followed by the edge side $r.fol1side$ (0=left, 1=right) of the object $r.fol1poin\uparrow$. For example the objects of Figure 4.4 give rise to the following values for these components:

Record	$r1$	$r2$	$r3$	$r4$	$r5$	$r6$	$r7$
$fol0poin\uparrow$	$r2$	$r3$	$r5$	$r3$	$r6$	$r7$	$r7$
$fol0side$	0	0	0	1	0	0	0
$fol1poin\uparrow$	$r1$	$r1$	$r4$	$r2$	$r4$	$r5$	$r6$
$fol1side$	0	1	0	1	1	1	1

Table 5.5. Edge-following in *restricted adjacency*

Note that if r is a block-continuation (i.e. if $r.ty = 2$), then $r.fol1poin$ is the object (block or block-continuation) of which r is the continuation.

5.2.4 Implementation

Let us now examine how *fol*0 and *fol*1 (in the case of *full adjacency*) or *fol*0*poin*, *fol*0*side*, *fol*1*poin* and *fol*1*side* (in the case of *restricted adjacency*) can be computed in real-time. The computation takes place in two of the subroutines invoked by procedure *allocate*, namely, *conbelow* and *newobject*, (see Appendix D, Sections D.2, and D.3). In fact, these are the subroutines that compute the adjacency relations at the object-level.

At this stage, it becomes virtually impossible to discuss our implementation without referring to the detailed code available in the appendices. Therefore, the reader might be well advised to defer the reading of this subsection until the time when he wishes to penetrate the darkest corner of the program.

In the first place, the implementation is discussed herafter in a form which is independent from the chosen adjacency option. Subsequently, *full* and *restricted adjacencies* will be considered separately.

OK, we assume that the reader has a copy of Appendix D handy. Recall from Chapter 3 that object parameters are going by simplified names. In particular, the variables *consu*, *conpr*, *lefpr*, etc, stand for $consu[i,u]$, $conpr[i,u]$, $lefpr[i,u]$, etc. We write *pnp* for $nripr[i,u-1]$. Moreover, there are several pointers of type *link*, namely, *z*, *p*, *last*, *s*, and *ss* which must be interpreted as follows:

- *z* points to the object to which procedure *conbelow* is applied.
- *p* points to a new object starting with $run[i,u]$.
- *last* points to the object containing $run[i,u-1]$.
- When $conpr[i,u] > 0$, we consider for $x = 0, \ldots, conpr[i,u]-1$ the xth object adjacent to $run[i,u]$ and above it. Then, *s* points to that object, while *ss* points to the $(x-1)$st object.

Now, one of the six cases of interest in Table 5.2 is treated in the procedure *conbelow*. The other five cases are handled in the procedure *newobject*.

(*i*) In *conbelow*, to the compound statement following IF $consu = 0$ THEN we must add the following information:

$$red(z\uparrow) \text{ follows } led(z\uparrow), \tag{5.8}$$

i.e., case ($A.1°$) holds for $led(z\uparrow)$.

(*ii*) In *newobject*, there are several additions:

(*a*) To the compound statement following IF $conpr = 0$ THEN we add the information that:

$$led(p\uparrow) \text{ follows } red(p\uparrow), \tag{5.9}$$

i.e., case ($D.1°$) holds for $red(p\uparrow)$.

(*b*) In the "IF $x = 0$ THEN" part, we add to the compound statement following the "ELSE" of the condition "IF $u > 0$ AND $lefpr < pnp$" the following information:

$$led(p\uparrow) \text{ follows } led(s\uparrow), \tag{5.10}$$

i.e., case ($A.2°$) holds for $led(s\uparrow)$.

(*c*) In the compound statement following

IF $x = conpr - 1$ THEN IF
$(xrec.objty = 1)$ OR $\big((xrec.objty = 0)$ AND $(xrec.rri = u)\big)$

(this statement means that $run[i, u]$ and $run[i, u + 1]$ are not adjacent to a common run above them), we add the following:

$$red(s\uparrow) \text{ follows } red(p\uparrow), \tag{5.11}$$

i.e., case ($D.2°$) holds for $red(p\uparrow)$.

(*d*) In the compound statement following the "ELSE" of the condition "IF $x = 0$" we get:

$$red(ss\uparrow) \text{ follows } led(s\uparrow), \tag{5.12}$$

i.e., case ($A.3°$) holds for $led(s\uparrow)$.

(*e*) In the case where $x = 0$ and $u > 0$ AND $lefpr < pnp$, which means that $run[i, u]$ and $run[i, u - 1]$ are adjacent to a common run above them, we have the following:

$$led(p\uparrow) \text{ follows } red(last\uparrow), \tag{5.13}$$

i.e., case ($D.3°$) holds for $red(last\uparrow)$.

Now, let us see how this can be implemented in both cases: *full* and *restricted adjacencies*.

In *full adjacency*, we have only to add in the compound statement corresponding to each case (i), $(ii.a)$, ..., $(ii.e)$ above the information conveyed by equations (5.8) to (5.13) in terms of the cases (1°), (2°) or (3°) of Table 5.2 with the fields $fol0$ and $fol1$ of the given record. Thus we write:

$$
\begin{aligned}
&z \uparrow .fol0 = 1 \text{ in case } (i).\\
&p \uparrow .fol1 = 1 \text{ in case } (ii.a).\\
&s \uparrow .fol0 = 2 \text{ in case } (ii.b).\\
&p \uparrow .fol1 = 2 \text{ in case } (ii.c).\\
&s \uparrow .fol0 = 3 \text{ in case } (ii.d).\\
&last \uparrow .fol1 = 3 \text{ in case } (ii.e).
\end{aligned} \tag{5.14}
$$

In *restricted adjacency*, there are more transformations. We delete from the two procedures *conbelow* and *newobject* all the statements concerning *precnnb*, *prefi*, *prela*, *preletori*, etc. We insert the contents of equations (5.8) to (5.13) in terms of the fields *fol0poin*, *fol0side*, etc, at the appropriate places. Thus we write:

$$
\begin{aligned}
&z \uparrow .fol0side = 1 \text{ and } z \uparrow .fol0poin = z \text{ in case } (i).\\
&p \uparrow .fol1side = 0 \text{ and } p \uparrow .fol1poin = p \text{ in case } (ii.a).\\
&s \uparrow .fol0side = 0 \text{ and } s \uparrow .fol0poin = p \text{ in case } (ii.b).\\
&p \uparrow .fol1side = 1 \text{ and } p \uparrow .fol1poin = s \text{ in case } (ii.c).\\
&s \uparrow .fol0side = 1 \text{ and } s \uparrow .fol0poin = ss \text{ in case } (ii.d).\\
&last \uparrow .fol1side = 0 \text{ and } last \uparrow .fol1poin = p \text{ in case } (ii.e).
\end{aligned} \tag{5.15}
$$

Note that with these transformations, the last ELSE in the body of *newobject* disappears, because it commands only one statement concerning a pointer *sucla*.

5.2.5 Comment

We have shown above how the records of type *objrec* can be adapted to make feasible the following of the cycles of the edge of F. We showed that the ten components of the object record describing the adjacency relations can be used in

order to follow these cycles. We showed also that they may be replaced by only four components, in which case the adjacency relations between objects are less explicitely detailed. Now several problems remain:

(1°) We must be able to determine when a cycle of the edge is completed.

(2°) We must be able to detect the relation between the different cycles of the edge of a given connected component.

(3°) We must be able to describe these cycles and the relations between them in a form that can be exploited by the user.

Problems (1°) and (2°) will be dealt with in Section 5.4, while problem (3°), which has the flavor of a theoretical prerequisite for the solution of the other two, is the subject matter of the next Section.

§5.3 The Neighborhood-Tree and its Representation

In this Section we examine the nature of the neighborhood and surrounding relations between connected components of F and B, and between cycles of the edge $\epsilon^+(F)$. We also show how these relations can be represented in an economical and practical way. The problem involves the topology of the plane, the dynamical coding of graphs, and some other theoretical problems on which we have to concentrate before returning to the description of the program.

The reader should take it just on faith right now that the procedure *allocate* can still be further modified in order to detect and describe in real-time the surrounding relations between the connected components of F and B. It is well-known that these relations form a tree. We will be confronted with a problem similar to the one which we encountered when we described the representation of the adjacency relations between the objects, namely, how to describe relations of unbounded degree with records of bounded size. (For instance, a connected component of F may have up to $M.N/4$ holes, but we must represent the link between that component and its holes with a small, constant number of parameters). We will show in this Section how the tree representing the surrounding relations between connected components of F and B can be represented as a string whose length is twice the size of that tree, and how this string can be represented with records and pointers.

It should be clear that, at this stage, the whole information contained in the neighborhood-tree is implicitely available. We now have to decide which part of it should be contained in the output. This question leads us to the second option alluded to in Section 5.1, *viz.*, the choice between *full* and *restricted surrounding*. Let us briefly examine this issue.

First, in any case we can delete the top vertex from the neighborhood-tree for it represents the connected component of B which contains the frame of G and surrounds all other connected components of F or B. Clearly, the information contained in that vertex is redundant. Moreover, if we were to retain it, no part of the neighborhood-tree could be output before the completion of the scan of the figure. When it is deleted, every remaining part of the tree can be output when the corresponding components have been fully disclosed, (see Section 6.1 for details). With that top vertex deleted, our approach offers the following alternatives:

- We retain the rest of the neighborhood-tree, in other words, all the surrounding relations between all connected components of F and all connected components of B, except the component of B which contains the frame of G. This is *full surrounding*.
- We delete from the neighborhood-tree all the edges occurring when a connected component of B surrounds one of F. In other words, connected components of F are taken in isolation; we retain only the surrounding relation between each connected component of F and its holes. This is *restricted surrounding*.

This Section is organized as follows: In 5.3.1, we recall some theoretical properties of binary pictures on a square grid, such as the neighborhood tree, the surrounding relation, etc. In 5.3.2, we show how the neighborhood tree can be described by either its vertex-string or its edge-string. We justify our choice of the edge-string as a representation. In 5.3.3, we concentrate on *full* and *restricted* surroundings. In 5.3.4, we outline our implementation of the edge-string approach.

5.3.1 The neighborhood-tree of a figure

Recall that the *frame* of grid G is the set of pixels $p(i,j)$ such that $j = 0$ or $N-1$, or $i = 0$ or $M-1$, and we make the frame assumption that the frame of G is included in B.

Given two disjoint subsets V and W of the grid G, we say that V *surrounds* W if every 4-path from W to the frame of G intersects V. (This corresponds to the definition of "4-surrounding" in [4]).

The frame assumption garantees that every cycle of the edge is closed. (This would also be true if we made the reverse assumption that the frame of G is included in F). A well-known result of digital topology (see for example [4],[5]) states the following:

> *For any k-connected component X of F and k'-connected component Y of B neighboring X, either X surrounds Y or Y surrounds X; moreover, there is at most one such Y neighboring X and surrounding it, and there is at most one such X neighboring Y and surrounding it.*

This property entails the following: If we draw the graph whose vertices (nodes) are all k-connected components of F and k'-connected components of B and whose edges join neighboring components, then that graph is a tree. This tree is called the *neighborhood-tree* or sometimes the *adjacency tree* of F. We can represent it as a descending tree as in Figure 5.6.b, where for two adjacent nodes, the component represented by the parent node surrounds the component represented by the successor node. The top vertex of the tree represents the connected component of B which contains the frame of G. As an example, Figure 5.6.b displays the neighborhood-tree of Figure 5.6.a.

We will now exhibit two correspondences between cycles of the edge $\epsilon^+(F)$ and edges of the neighborhood-tree, and between vertices and edges of that tree. First, a cycle of $\epsilon^+(F)$ is of the form $\epsilon^+(X,Y)$, and it corresponds to the pair $\{X,Y\}$ of adjacent vertices of the neighborhood-tree, in other words, to the edge joining them. Thus, cycles of the edge $\epsilon^+(F)$ correspond to edges of the neighborhood-tree. Second, to every edge of the neighborhood-tree there corresponds one vertex of that tree, which we choose to be the bottom vertex of that edge. By this we define a one-to-one correspondence between the set of edges and the set of all vertices of that tree, except the top vertex.

Next, if we combine these correspondences, we get a new correspondence between cycles of $\epsilon^+(F)$ and connected components of F and B: The cycle $\epsilon^+(X,Y)$ corresponds to Y if X surrounds Y, and to X if Y surrounds X.

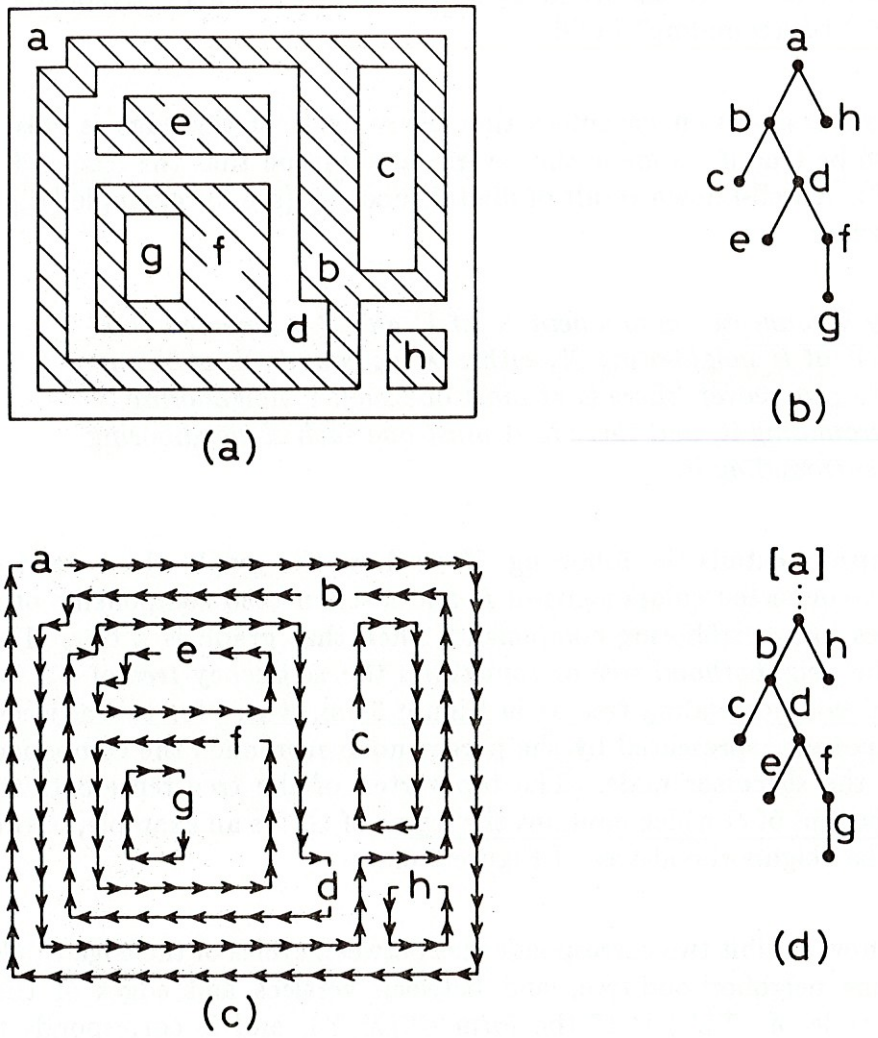


Figure 5.6. Neighborhood-trees representing connected components (a,b) and cycles (c,d).

We have thus a one-to-one correspondence between the cycles of $\epsilon^+(F)$ and all connected components of F and B, except for the connected component of B which contains the frame. To such a connected component corresponds the

outer cycle of its edge.

Now, we can extend this correspondence, in order to include in it the connected component of B which contains the frame, by drawing an edge of the grid along the frame, and declaring that it belongs to the edge $\epsilon^+(F)$. It is the outer cycle of the edge of the connected component of B which contains the frame. Then, the correspondence is complete. It is illustrated in Figures 5.6.c and 5.6.d, where we show how the labels of the connected components (and of the vertices of the tree) can be used for the cycles of $\epsilon^+(F)$ and for the edges of the tree.

This correspondence between vertices and edges, between connected components and cycles will be used in Subsection 5.3.2 for the purpose of defining the *vertex-string* and the *edge-string* which represent the adjacency-tree.

5.3.2 The vertex-string and the edge-string

It goes without saying that it is desirable to represent the neighborhood tree of F in an economical way. Indeed, it would be unwise to represent the surrounding relations by giving for each component an array whose entries point to its holes, since the number of holes in a connected component may be very large, while arrays have a bounded size. This problem is similar to the one which we encountered in Section 4.3, when we proposed a representation for the adjacency relations between objects. We opted for a dynamical solution, with transversal pointers $pre(suc)letori(ritole)$ scanning the sequence of objects adjacent to a given object. This allowed us to represent the adjacency relations with a fixed number of object parameters.

Here, we will also adopt a dynamical solution. We will show that the neighborhood-tree can be represented as a string. This provides a concrete solution for the representation of cycles and surrounding relations: Cycles will be represented by records, and strings will be represented by double chain of pointers between these records.

Let us next examine two possible string representations for the neighborhood-tree, namely, the *vertex-string* and the *edge-string*.The vertex-string is explicitly defined in [1], though under a different name, while the edge-string is implicitely used in [5].

Let $\mathcal{T}$ be the neighborhood-tree of a figure and let t be its top vertex. Let P be a path in $\mathcal{T}$ such that:

(i) P begins and ends in t.

(ii) P passes through every vertex of $\mathcal{T}$ at least once (and so P passes through every edge of $\mathcal{T}$ at least once).

(iii) P is shortest possible with respect to (i) and (ii).

If V is the sequence of vertices of $\mathcal{T}$ in P, then the string 'V' is the *vertex-string* of $\mathcal{T}$. Now the names of the vertices of $\mathcal{T}$ (except t) can be used to label the edges of $\mathcal{T}$, as explained above. If E is the sequence of edges of $\mathcal{T}$ in P (labelled with the names of the corresponding vertices of $\mathcal{T}$), then 'tEt' is the *edge-string* of $\mathcal{T}$. We add t before and after E, because t is the only vertex which does not appear in E.

Clearly, as P is not uniquely determined, the vertex and edge-strings of $\mathcal{T}$ are not uniquely defined. In fact, the different forms that they may take correspond to the different geometrical representations of the tree $\mathcal{T}$ in the plane.

The strings which correspond to the neighborhood-trees in Figure 5.6 are

$$'a\frown b\frown c\frown b\frown d\frown e\frown d\frown f\frown g\frown f\frown d\frown b\frown a\frown h\frown a'$$

for the vertex-string, and

$$'a\frown b\frown c\frown c\frown d\frown e\frown e\frown f\frown g\frown g\frown f\frown d\frown b\frown h\frown h\frown a'$$

for the edge-string.

Now, the path P can be visualized by a 4-path Q in G as follows: Q begins and ends in the frame of G and passes through every connected component of F and B a minimal number of times. Then the vertex-string is the sequence of connected components of F and B in Q, while the edge-string is the sequence of cycles of $\epsilon^+(F)$ which are crossed by Q (where we suppose that the edge of G is labelled t and is crossed by Q).

Let us now give some elementary properties of these two strings. Write $V(\mathcal{T})$ for the number of vertices of $\mathcal{T}$, and for any vertex x of $\mathcal{T}$, let $v(x)$ be the number of (adjacent) successors of x in $\mathcal{T}$. For any two vertices x and y of $\mathcal{T}$, we will say that x is *below* y if there is a path in $\mathcal{T}$ descending from y to x. Then, we have the following properties:

(*i*) The vertex and edge-strings have respective lengths $2V(\mathcal{T})-1$ and $2V(\mathcal{T})$.

(*ii*) The vertex x appears $v(x)+1$ times in the vertex string and 2 times in the edge-string.

(*iii*) In the *vertex-string*, two adjacent labels correspond to two neighboring connected components of F and B and *vice-versa*. On the other hand in the *edge-string* two neighboring labels correspond to two cycles of $\epsilon^+(F)$ which are separated by only one connected component of F or B (and therefore to two neighboring connected components of F and B), *but the converse is not true.*

(*iv*) Let x and y be two vertices of $\mathcal{T}$. Then the following holds for both the *vertex* and *edge-strings*:

(*iv.a*) If x is below y, then all occurrences of x in the string are enclosed by two successive occurrences of y in that string.

(*iv.b*) If x is not below y, then no occurrence of x in the string is enclosed by two occurrences of y in that string.

Property (*iv*) is quite important from our present viewpoint. It really means that the vertex and edge-strings contain all the information conveyed by $\mathcal{T}$. Thus any of these strings can be used as a representation for the neighborhood-tree.

Let us briefly outline how the edge-string can be derived from the vertex string. Consider the succession of pairs of consecutive vertices in the vertex-string. Every such pair corresponds to an edge in $\mathcal{T}$. Then replace every such pair by the vertex it contains which is below the other. Then by adding t at the beginning and end of the resulting string, one gets the edge-string. With the example of Figure 5.6, we get the following derivation:

vertex-string

edge-string

A simple, iterative method for constructing the vertex and edge-strings from the neighborhood-tree can be described as follows:

(1°) A tree containing only one vertex, say v, has vertex-string 'v' and edge-string '$v\ v$'.

(2°) Let $\mathcal{T}$ be a tree having vertex-string '$V_{\mathcal{T}}$' and edge string '$E_{\mathcal{T}}$'. Suppose that we attach to a vertex v of $\mathcal{T}$ (and below it) another tree $\mathcal{S}$ having vertex-string '$V_{\mathcal{S}}$' and edge string '$E_{\mathcal{S}}$'. Let $\mathcal{R}$ designate the resulting tree. Then, its vertex and edge-strings '$V_{\mathcal{R}}$' and '$E_{\mathcal{R}}$' can be constructed by applying the following rules:

'$V_{\mathcal{R}}$' is obtained from '$V_{\mathcal{T}}$' by replacing in it the first occurrence of 'v' by '$v\ V_{\mathcal{S}}\ v$'.

'$E_{\mathcal{R}}$' is obtained from '$E_{\mathcal{T}}$' by replacing in it the first occurrence of 'v' by '$v\ E_{\mathcal{R}}$'.

These are the main properties of the vertex and edge-strings which are relevant to our present concerns. Either are about equally convenient for the purposes of representating surrounding relations in an image. A choice has been made for the edge-string for the following three reasons:

(*i*) The vertex-string corresponds to the connected components of F and B, while the edge-string corresponds to cycles of $\epsilon(F)$. As we will ultimately recognize connected components by an analysis of cycles, the edge-string proves more convenient.

(*ii*) In the vertex-string, the number of occurrences of a vertex v is variable, while in the edge-string, each edge occurs exactly two times. This facilitates the representation of these occurrences with static structures such as Pascal records.

(*iii*) In the edge-string there is a convenient way to number the occurrences of edges. For a label v corresponding to the outer edge of a connected component of F (or to that component), the first occurrence of v is numbered 0, while the second is numbered 1. On the other hand, if v corresponds to the outer edge of a connected component of B (or to that component), then we do the contrary: the first occurrence of v is numbered 1, while the second one is numbered 0. Then the sequence of numberings in the edge-string is 10. . 10. For instance, the edge-string corresponding to Figure 5.6.d becomes:

'$a1\ b0\ c1\ c0\ d1\ e0\ e1\ f0\ g1\ g0\ f1\ d0\ b1\ h0\ h1\ a0$'

It is readily seen that this numbering has the following topological meaning: Consider the path Q defined above. The occurrences of the edges in the string correspond to their crossings with Q. Then an occurrence numbered 0 corresponds to a transition of Q from white to black, while an occurrence numbered 1 corresponds to a transition of Q from black to white. We will see later on that when we use row i as a path crossing currently unclosed cycles, the occurrences numbered 0 will correspond to *left* edges of runs, while occurrences numbered 1 will correspond to *right* edges of runs.

We are now ready to consider the possible restrictions in the neighborhood tree, i.e. the matter of *full* and *restricted surroundings.*

5.3.3 Full surrounding and restricted surrounding

In the process of detecting connected components of a figure, one does not always need to extract all surrounding relations. For instance, if the input is a typed text, we must merely isolate the characters. Thus, one can make a choice of which parts of the neighborhood-tree should be recognized and written in the output.

However, we must, in any case, retain a minimum of the information contained in that tree: We will recognize the completion of a connected component by the closure of the outer cycle of its edge; we will also get the various objects making up that component by following all the cycles of its edge. Therefore, it will be necessary to store, at least, the link between the outer cycle and the inner cycles of the edge of each connected component of F, in other words, between each connected component and its holes. We show in Figure 5.7 the neighborhood-tree of Figure 5.6.b and the portion of it which represents the surrounding relations between connected components of the figure and their respective holes. The choice to use that minimum of information from the neighborhood-tree will be called *restricted surrounding.* The opposite choice, which consists in retaining the whole information contained in the neighborhood-tree, will be called *full surrounding.* There may be other intermediate choices but they will not be considered here.

A little tought reveals that—even if this sounds paradoxical at first sight—the most practical way to retain the whole information in the neighborhood-tree is to start by deleting the top vertex from that tree. Indeed, suppose that we were scanning an image in a single raster scan, and detecting the connected components of the figure together with the complete neighborhood-tree. The connected component Y of the background which contains the frame, and which

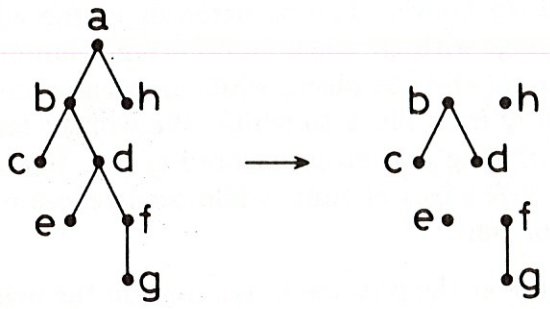


Figure 5.7. The neighborhood-tree of Figure 5.6.a.

corresponds to the top vertex t of the neighborhood-tree $\mathcal{T}$, would be completely disclosed only at the end of the scanning. In particular, it is only at that time that we could ascertain how many vertices of $\mathcal{T}$ are adjcent to t. Thus, if we were to keep $\mathcal{T}$ in its original form, we should have to wait until the end of the scan before being in a position to write any information in the output. This would make it impossible to process the image in real-time.

Conversely, suppose that vertex t is deleted from $\mathcal{T}$. Then, all the information contained in $\mathcal{T}$ is also contained in $\mathcal{T}\backslash\{t\}$, because it is known beforehand (by the frame assumption) that t and the corresponding connected component Y of the background always exist. In fact, $\mathcal{T}\backslash\{t\}$ is a union of disjoint trees, say $\mathcal{T}^1, \ldots, \mathcal{T}^r$, and if the top vertex t_s of each $\mathcal{T}^s$ $(s = 1, \ldots r)$ is known, then $\mathcal{T}$ can be reconstructed. In this way, we can meet our real-time requirement: At the time when a connected component X of the figure is completely processed, it is possible to determine whether it neighbors Y, in which case it is possible to write into the output the contents of the tree $\mathcal{T}^s$ corresponding to X. We will thus constantly assume that the top vertex of the neighborhood-tree, in other words, the connected component of the background containing the frame, is not taken into consideration even in *full surrounding*. The top vertex of each $\mathcal{T}^s$ is said to be *maximal for surrounding*

Our program offers the choice between full and restricted surrounding, in the same way that it offers the choice between full and restricted adjacencies. There are no other major options except for the choices of the constants k, *blen*, *clen*, etc.

Let us now examine the edge-string corresponding to the truncated form of the neighborhood-tree in full and restricted surrounding.

In *full surrounding*, we consider the graph $\mathcal{T}_1 = \mathcal{T} \backslash \{t\}$, which is a union of disjoint trees. Each one of them can be described by an edge string. Then, the global edge-string of $\mathcal{T}_1$ is, in fact, the set of edge-strings of its connected components. For example, the edge string of Figure 5.6.c becomes:

$$\text{`}b \frown c \frown c \frown d \frown e \frown e \frown f \frown g \frown g \frown f \frown d \frown b\text{'}, \text{ and } \text{`}h \frown h\text{'}$$

and, with the numbering of the occurrences, we get

$$\text{`}b0 \frown c1 \frown c0 \frown d1 \frown e0 \frown e1 \frown f0 \frown g1 \frown g0 \frown f1 \frown d0 \frown b1\text{'}, \text{ and } \text{`}h0 \frown h1\text{'}.$$

In *restricted surrounding*, a new graph $\mathcal{T}_2$ is obtained from $\mathcal{T}_1$ by deleting from it all edges corresponding to the cases where a connected component of B surrounds one of F. For instance, Figure 5.7 displays the tree $\mathcal{T}_2$ corresponding to the tree $\mathcal{T}$ of Figure 5.6.b. Accordingly, we get the following set of edge-strings:

$$\text{`}b \frown c \frown c \frown d \frown d \frown b\text{'}, \quad \text{`}e \frown e\text{'}, \quad \text{`}f \frown g \frown g \frown f\text{'}, \text{ and } \text{`}h \frown h\text{'}$$

or, with the numbering of the occurrences,

$$\text{`}b0 \frown c1 \frown c0 \frown d1 \frown d0 \frown b1\text{'}, \quad \text{`}e0 \frown e1\text{'}, \quad \text{`}f0 \frown g1 \frown g0 \frown f1\text{'}, \text{ and } \text{`}h0 \frown h1\text{'}.$$

Let us note a particular property of the edge-strings in *restricted surrounding*: The string corresponding to a connected component X of F having, say, r holes takes the form

$$\text{`}x \frown y(1) \frown y(1) \frown \ldots \frown y(r) \frown y(r) \frown x\text{'},$$

where $y(1) \frown \ldots \frown y(r)$ correspond to the r holes. This string can be represented by a cycle

$$(x, y(1), \ldots, y(r)), \tag{5.16}$$

provided that one records simultaneously that x corresponds to the outer cycle of the edge $\epsilon^+(X)$ of connected component X. We call that string the *simplified edge-string*. We will use it in the following subsection for representing the cycles and surrounding relations in restricted surrounding.

5.3.4 The representation of cycles and edge-strings

A few words are in order about the data structure used to store the description of cycles and edge-strings in both *full* and *restricted surroundings*. The data structure should meet the following requirement:

(*i*) For every cycle, given an object along it, we need an access to the (left or right) edge of that object—which belongs to that cycle—in order to be able to follow the succession of edge-elements in that cycle.

(*ii*) We need to know if a given cycle is the outer cycle of the edge of a connected component of F or of B. This determines the numbering of the occurrences of that edge in the edge string.

(*iii*) In *full surrounding*, we need to identify the top vertex of each tree in the truncated neighborhood-tree $\mathcal{T}_1$.

(*iv*) Given any cycle, we need to know those cycles which precede and follow each of its two occurrences in the edge-string. This permits us to reconsruct the edge-string whether we traverse it from left to right or from right to left.

We will represent cycles by records, and strings by double chains of pointers to records corresponding with cycles. The type corresponding to cycles' records is called *cyrec*. We define *cypoin* $=\uparrow$ *cyrec*, in other words *cypoin* is the type of pointers pointing to variables of type *cyrec*. Now let us examine how the requirements (*i*) to (*iv*) can be met.

(*i*) Records of type *cyrec* contain a component *acces* of type *link*, pointing to a variable of type *objrec*, such that if c is the record of a given cycle, then $(c.acces)\uparrow$ is an object whose left edge belongs to that cycle.

(*ii*) We define next the component *whi* (standing for white) of type 0..1 such that for a given cycle record c, $c.whi = 0$ if and only if c corresponds to the outer edge of a connected component of F. Note that $c.whi$ is the number of the first occurrence of c in the edge-string.

(*iii*) In *full surrounding*, the identification of the top vertex of each tree will take place in the procedure *closechain* which is discussed in Section 5.3.5.

(*iv*) We define four pointers $pr0$, $pr1$, $pl0$, $pl1$ of type *cypoin*, as follows: Let c_{l0} and c_{r0} be the names of the records corresponding to the two cycles which are in the edge-string respectively the left and the right neighbors of the occurrence numbered 0 of a cycle whose record is c, similarly let c_{l1} and c_{r1} be the ones corresponding to the occurrence numbered 1. Then, the role of these pointers is best illustrated as follows:

$$\smile c_{l0} \xleftarrow{c.pl0} c0 \xrightarrow{c.pr0} c_{r0} \smile \cdots \smile c_{l1} \xleftarrow{c.pl1} c1 \xrightarrow{c.pr1} c_{r1} \smile$$

If c represents the top vertex of a connected component $\mathcal{T}^s$ of $\mathcal{T}_1$, then, according to the above definitions, the two pointers $c.pl0$ and $c.pl1$ have no destination. They may be set to NIL, or they may even be left undefined.

In the case of *restricted surrounding*, we can use the *simplified edge-string* of (5.16). We can use only two pointers, namely, pl and pr pointing respectively to the left and right neighbor of the single occurrence of the cycle in the *simplified edge-string*. (Here pl and pr are always defined $\neq NIL$). Note also that we have:

$$\begin{aligned} pl &= pl1; \\ pr &= pr0. \end{aligned} \tag{5.17}$$

The following table gives an example of the pointers $pl0$, $pl1$, $pr0$ and $pr1$ for the edge-string corresponding to Figure 5.6.

Record	a	b	c	d	e	f	g	h
$pl0$	$\rightarrow h$	$\rightarrow a$	$\rightarrow c$	$\rightarrow f$	$\rightarrow d$	$\rightarrow e$	$\rightarrow g$	$\rightarrow b$
$pl1$	NIL	$\rightarrow d$	$\rightarrow b$	$\rightarrow c$	$\rightarrow e$	$\rightarrow g$	$\rightarrow f$	$\rightarrow h$
$pr0$	NIL	$\rightarrow c$	$\rightarrow d$	$\rightarrow b$	$\rightarrow e$	$\rightarrow g$	$\rightarrow f$	$\rightarrow h$
$pr1$	$\rightarrow b$	$\rightarrow h$	$\rightarrow c$	$\rightarrow e$	$\rightarrow f$	$\rightarrow d$	$\rightarrow g$	$\rightarrow a$

Table 5.6. Following the cycles in Figure 5.6

From what we just said it is clear that, *in restricted surrounding*, the *cyrec* records use some of the components which were defined in *full surrounding*. This situation differs from the choice of *full* and *restricted adjacencies* where the pointers in *objrec* records were completely different in each case.

Now that we have resolved the problem of representing the surrounding relations between connected components of F and B, we will consider that of detecting the closure of cycles and the construction of the *cyrec* records.

§5.4 The Construction of Cycles and Edge-Strings

5.4.1 Introduction

In the preceding two sections, we made it clear that real-time detection of connected components can be achieved by following edge cycles and detecting their closure. When some outer cycle gets closed, one connected component is completely disclosed. It can be found by following its outer cycle and the inner cycles—if any—directly surrounded by it. We also demonstrated that surrounding relations can be represented by edge-strings which can be coded by records and pointers.

However, at this stage, we are still confronted with the important problem of extending the procedure *allocate* in order to include in it both the data structure representing the edge-string, and the mechanism to detect the closure of cycles. To clarify the situation, let us examine the various possibilities that can occur when some, arbitrary row is being scanned. As the row is traversed, we assume that unclosed cycles, called *chains* are encountered. Presently, our task is to lay bare the surrounding relations between chains and between chains and cycles in the portion of the image that has been scanned thus far.

As the scan is progressing, four possible operations on chains may have to be performed:

- A chain can be enlarged.
- A new chain can be created.
- Two existing chains can be merged into one larger chain.
- A chain can be closed into a cycle.

These four operations are executed by four new procedures nested inside *allocate*, namely, *extendchain*, *newchain*, *mergechain*, and *closechain*. Two additional procedures, *enclose* and *concatenate*, take care of ancillary details and are

used in *mergechain* and *closechain*. Out of these six new procedures, *closechain* plays the prominent role. It activates the creation of *cyrec* records, as well as the output procedure.

With this as a preamble, let us examine the behavior of chains from the viewpoint of their neighborhood relations.

5.4.2 Relations between chains and cycles

For the purposes of the present discussion, let us assume that the scan has reached the end of row i in Figure 5.8.a. Clearly, every chain begins and ends on the right and left edges of some runs on row i respectively. If s is the name of such a chain, we can label its beginning $s1$, and its end $s0$. Our motivation behind this choice is twofold.

(*i*) The numbers 0 and 1 correspond respectively to left and right sides of run edges.

(*ii*) As row i is scanned from left to right, this numbering is consistent with the theory in Section 5.3.2.

Figure 5.8.c displays the (partial) neighborhood-tree corresponding to Figure 5.8.a. Marked edges correspond to chains; unmarked ones correspond to cycles. This tree conveys various kinds of information, some of which is relevant to our present concerns, some of which isn't. Let us distinguish first the following three kinds of information.

(*i*) Given several chains or cycles, the tree tells us whether they are included either in the edge $\epsilon^+(X)$ of some connected component X of F or in the edge $\epsilon^-(Y)$ of some connected component Y of B. This implies that there exists an edge-string representation in which any two of these chains or cycles are adjacent. For example, in Figure 5.8.a, this relationship applies to

- u and v (through B),
- a, c, d, e and u (through F),
- g and h (through B),
- w, x, f, g and b (through F),
- a, b, y, z and t (through B).

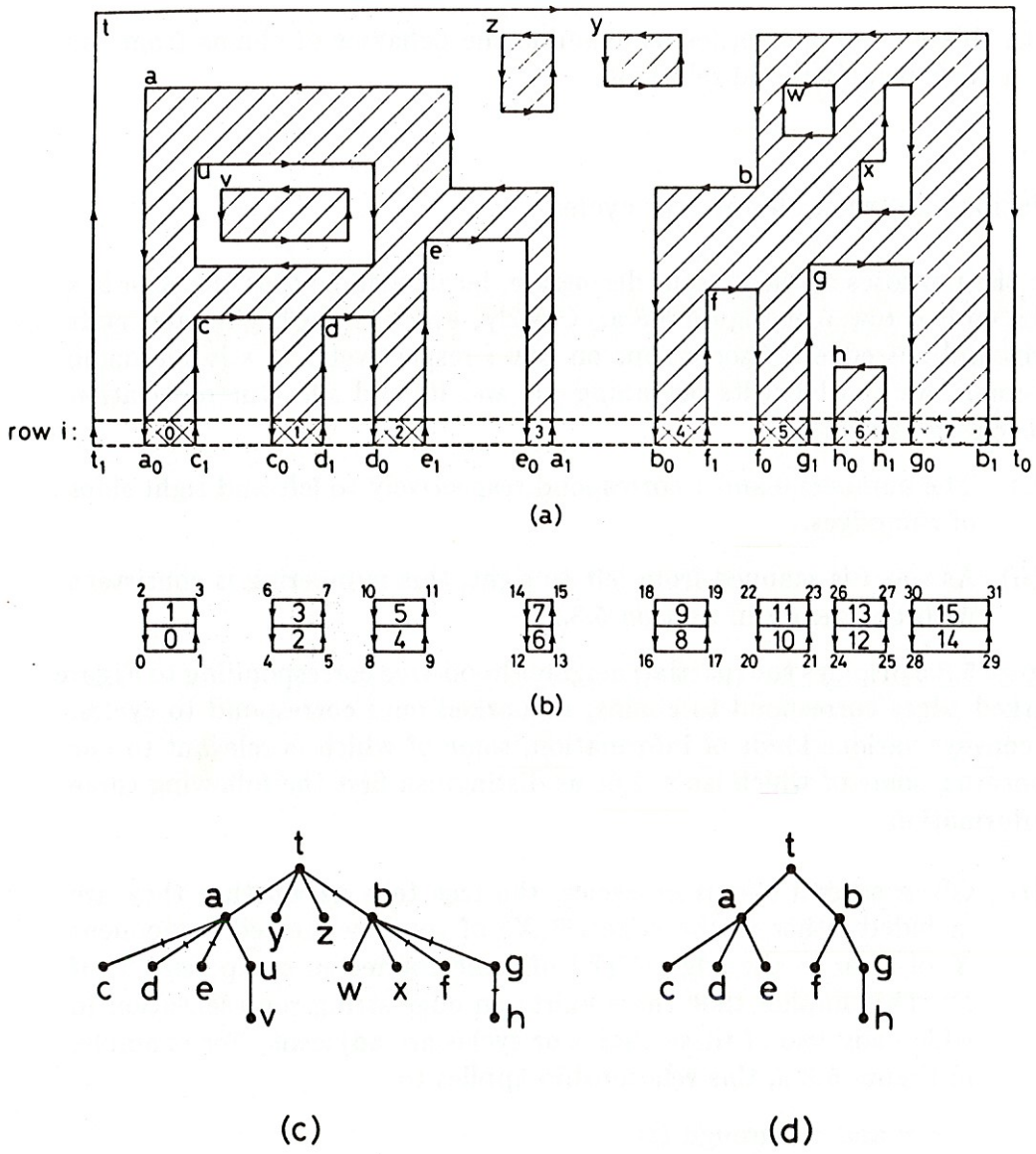


Figure 5.8.
(a) Chains and cycles in a portion of a figure.
(b) Labelling of runs and chain extremities.
(c) Neighborhood-tree for the partial figure.
(d) Neighborhood-tree with cycles removed.

(*ii*) A chain or a cycle whose white side is k'-connected to the frame of G is maximal for surrounding in the portion of the image that has been scanned. For instance, this is the case with a, b, y and z (with t) in Figure 5.8.a.

(*iii*) A cycle never surrounds a chain. But, surrounding relations are possible between cycles as is exemplified by cycles u and v in Figure 5.8.a.

The kind of information which is expressed by (*i*)–(*iii*) is relevant to our present concerns for the very reason that it is invariant in the course of the processing. At this point, it is an easy matter to devise examples of information conveyed by the partial neighborhood-tree which may fail to be invariant. For instance, according to Figure 5.8.c, a "seems" to surround c, d and e; however, in the course of the processing of subsequent rows, some or all of these chains can be merged into one or more cycles, and some or all of these surrounding relations may vanish.

Let us see how the information involved in (*i*)–(*iii*) above can be recorded, first in general, then in both full and restricted surroundings.

In the first place, we note that the restriction of (*i*) to chains is determined by the sequence of beginning and ends of all chains along row i. To see this, we first delete from the tree in Figure 5.8.c all the vertices corresponding to cycles, thereby obtaining the tree of Figure 5.8.d. Then, it is readily verified that the edge-string of that tree yields the sequence of beginnings and ends of chains encountered when scanning row i from left to right. Thus, it is easy to see that the information involved in (*i*) can be recorded by performing the following two operations:

(*a*) We keep record of the sequence of chains extremities.

(*b*) For every connected component Z of F or B whose edge contains at least one chain, we gather together into an edge-string, all the cycles of $\epsilon^+(F)$ which are in the edge of Z as well as all the cycles surrounded by them. For instance, in Figure 5.8.a, we get the strings

$$S_1 = u1\frown v0\frown v1\frown u0,$$
$$S_2 = y0\frown y1\frown z0\frown z1,$$
$$S_3 = w1\frown w0\frown x1\frown x0.$$

Next, for each of these strings, we choose a chain in the edge of Z and we associate that string with the extremity (beginning or end) whose left side is in Z. Readily, if Z is in F, we choose the beginning of the chain; if Z is in B, we choose the end of the chain. In this way,

S_1 may be associated with either of $a1$, $c1$, $d1$ and $e1$.
S_2 may be associated with either of $a0$, $b0$, and $t0$.
S_3 may be associated with either of $b1$, $f1$, and $g1$.

We note that (*b*) simultaneously records the information involved in (*iii*). Eventually, the information involved in (*ii*) can be recorded by attaching to each chain a number equal to one if the chain's white side is k'-connected in B to the frame of G, and equal to zero otherwise. We call that number the *maximality number* of the chain.

Let us now examine the specializations of this approach corresponding to full and restricted surrounding respectively.

Full surrounding

Recall that, in full surrounding, we merely ignore the top vertex t of $\mathcal{T}$. Thus, chains of $\epsilon^+(F)$ corresponding to top vertices of $\mathcal{T}_1 = \mathcal{T}\backslash\{t\}$ can be output together with the cycles they surround as soon as they are closed into cycles. Thus, in the situation depicted by Figure 5.8.a, cycles y and z were output at some previous stage and no string must be attached to $a0$ and $b0$. More generally, we have the following rule:

No edge-string can be attached to a chain end s0 having maximality number one.

The situation is completely described by giving:

(1°) The pairs of chains extremities,
(2°) the maximality number of each chain,
(3°) the string, if any, attached to each chain extremity.

Restricted surrounding

Recall that, in restricted surrounding, we are interested in only the surrounding relations—if any—between the outer and inner cycles of the edge $\epsilon^+(X)$ of a connected component X of F. Presently, the consequences are twofold. On

the first hand, the maximality number becomes evidently useless. On the other hand, edge-strings can be attached only to chain beginnings. Indeed, the outer cycle r of some $\epsilon^+(X)$ can be output as soon as detected and need never be attached to any chain end $s0$. For instance, in Figure 5.8, the string $u1\smile u0$ (or the simplified string (u)) can be associated with either of $a1$, $c1$, $d1$ and $e1$, whereas the string $w1\smile w0\smile x1\smile x0$ (or the simplified string (w,x)) can be associated with either of $b1$, $f1$ and $g1$.

Thus, the situation is completely described in this case by giving:

(1°) The pairs of chains extremities,
(2°) the simplified string, if any, attached to each chain beginning.

We are now ready to turn to the consideration of the implementation of this approach. This is the subject matter of the next two sections. To this end, we will need to lift the temporary assumption that the scan is at the end of row i. This will oblige us to consider chain extremities on both row $i-1$ and row i.

5.4.3 Representation of chains extremities and their properties

Let us assume that the image has been scanned up to $run[i,u]$ included. In this situation, chains can begin and end on rows i or $i-1$. Recall that every chain begins in the right edge of some run while it ends in the left edge of some run. Recall also that the constant $maxnbr$ is an upper bound to the number of runs on a row. Therefore, $2 \times maxnbr$ and $4 \times maxnbr$ provide upper bounds to the number of runs on two adjacent rows and the number of their left and right edges respectively. Therefore, we define four integer subranges:

$$\begin{aligned} max1 &= 0..maxnbr-1; \\ max2 &= 0..2\times maxnbr-1; \\ max4 &= 0..4\times maxnbr-1; \\ m2 &= -1..2\times maxnbr-1. \end{aligned} \tag{5.18}$$

Integers in the subranges $max1$, $max2$ and $max4$ are used to number respectively the runs on row i, the runs on rows i and $i-1$, and the chains extremities. Integers in the subrange $m2$ are used to number either the runs on rows i and $i-1$ or the absence of runs.

We now introduce various numberings. Given some w in $max1$, we label

$$\begin{aligned} &run[i, w] \text{ by } 2 \times w, \\ &run[i-1, w] \text{ by } 2 \times w + 1. \end{aligned} \tag{5.19}$$

Now, if a run is labelled by some x in $max2$, then we label

$$\begin{aligned} &\text{its left edge by } 2 \times x \\ &\text{it right edge by } 2 \times x + 1. \end{aligned} \tag{5.20}$$

In other words, for any w in $max1$ we label

$$\begin{aligned} led[i, w] \text{ by } y &:= 4 \times w, \\ red[i, w] \text{ by } y &:= 4 \times w + 1, \\ led[i-1, w] \text{ by } y &:= 4 \times w + 2, \\ red[i-1, w] \text{ by } y &:= 4 \times w + 3, \end{aligned} \tag{5.21}$$

where y is in $max4$. This labelling of runs and chain extremities is illustrated in Figure 5.8.b.

Let us next examine how we code the pairs of chain extremities. To this end we define the array

```
chex : ARRAY[max4] OF m2
```

where *chex* stands for chain extremities.

Given some run-edge numbered y in $max4$ such that y is neither the beginning nor the end of a chain, we set

$$chex[y] := -1. \tag{5.22}$$

Given the two extremities y and z (in $max4$) of a chain, we set

$$\begin{aligned} chex[y] &:= \lfloor z/2 \rfloor, \\ chex[z] &:= \lfloor y/2 \rfloor. \end{aligned} \tag{5.23}$$

One may wonder why we define $chex[y]$ and $chex[z]$ as $\lfloor z/2 \rfloor$ and $\lfloor y/2 \rfloor$ respectively. The reason is that there is a kind of correlation between y and z in that they have opposite parities. It turns out that $chex[y]$ is the number of the run containing the opposite end of the chain containing y. Note also that the array *chex* allows us to determine a chain extremitiy from its opposite extremity. Indeed, (5.22) and (5.23) imply that for any y in $max4$, if $chex[y] \neq -1$, then y is a chain extremity, and the opposite end of that chain is

$$z := \begin{cases} 2 \times chex[y] + (1 - y\text{MOD}2), & \\ 2 \times chex[y] & \text{if } y \text{ is odd,} \\ 2 \times chex[y] + 1 & \text{if } y \text{ is even.} \end{cases} \tag{5.24}$$

For instance, in Figure 5.8.a, we have $maxnbr \geq 8$ and we get the following values for *chex*:

$$\left.\begin{matrix} chex[0]=6 \\ chex[13]=0 \end{matrix}\right\} a \text{ begins in 13 and ends in 0,}$$

$$\left.\begin{matrix} chex[1]=2 \\ chex[4]=0 \end{matrix}\right\} c \text{ begins in 1 and ends in 4,}$$

$$\left.\begin{matrix} chex[5]=4 \\ chex[8]=2 \end{matrix}\right\} d \text{ begins in 5 and ends in 8,}$$

$$\left.\begin{matrix} chex[9]=6 \\ chex[12]=4 \end{matrix}\right\} e \text{ begins in 9 and ends in 12,}$$

$$\left.\begin{matrix} chex[29]=8 \\ chex[16]=14 \end{matrix}\right\} b \text{ begins in 29 and ends in 16,} \tag{5.25}$$

$$\left.\begin{matrix} chex[17]=10 \\ chex[20]=8 \end{matrix}\right\} f \text{ begins in 17 and ends in 20,}$$

$$\left.\begin{matrix} chex[21]=14 \\ chex[28]=10 \end{matrix}\right\} g \text{ begins in 21 and ends in 28,}$$

$$\left.\begin{matrix} chex[25]=12 \\ chex[24]=12 \end{matrix}\right\} h \text{ begins in 25 and ends in 24,}$$

and $chex[m] = -1$ for any other value of m in $max4$.

Aside from chain extremities, we have to define the data structure required to accommodate maximality numbers of chains (in full surrounding), and edge-strings attached to chain extremities.

In *full surrounding* maximality numbers of chains are stored in the array

```
sm : ARRAY[max4] OF binary
```

where for every y in $max4$, $sm[y]$ is set equal to the maximality number of the chain containing y. For instance, in Figure 5.8.a, chains a and b have maximality number one. There follows that:

$$\begin{aligned} sm[y] = 1 \text{ for } \quad y = & \ 0,\ 2,\ 13,\ 15, \\ & 16,\ 18,\ 29,\ 31, \\ = 0 \text{ otherwise,} & \end{aligned}$$

because the edges numbered 0, 2, 13, and 15 are in chain a, while those numbered 16, 18, 29 and 31 are in chain b.

Let us consider next how to associate strings of cycles to chain extremities. We already know that a string can be coded by a double chain of pointers linking the records corresponding to the cycles of that string. This double chain allows us to scan the string from left to right and from right to left. Thus, when we wish to associate a string to some y in $max4$, it merely suffices to attach to y two new pointers pointing to the records correponding to the first and last cycles in the string. To this end, we define two arrays

```
becs, encs : ARRAY[max4] OF cypoin
```

If a string $u \ldots v$ is associated to the chain extremity y, then, $becs[y]$ points to u, and $encs[y]$ points to v. The names *becs* and *encs* are mnemonics for beginning and end of the corresponding string.

In 5.4.2, we established the rule stating that, in *full surrounding* no edge-string can be attached to a chain end $s0$ having maximality number one. This rule can now be translated as follows:

If y is even and $sm[y] = 1$, then $becs[y] = encs[y] = NIL$.

Note also that for any z in $max4$, $becs[z] = NIL$ if and only if $encs[z] = NIL$. Furthermore, $chex[z] = -1$ implies that $becs[z] = encs[z] = NIL$.

The example of Figure 5.8.a may give the following assignments:

- *becs*[13] and *encs*[13] pointing both to u, *i.e.*, string $u1\_v0\_v1\_u0$ being associated to $a1$.
- *becs*[29] and *encs*[29] pointing to w and x respectively, *i.e.*, $w1\_w0\_x1\_x0$ being associated to $b1$.
- $becs[m] = encs[m] = NIL$ for any other value of m in $max4$.

In *restricted surrounding*, we have no maximality numbers, and we associate strings to beginning of chains only. We could proceed as above using the arrays *becs* and *encs* with the restriction that $becs[y] = encs[y] = NIL$ for any even y. However, we can gain somewhat on storage space by defining instead two arrays

```
beho, enho : ARRAY[max2] OF cypoin
```

where for any y in $max2$, $beho[y]$ and $enho[y]$ correspond to $becs[2 \times y + 1]$ and $encs[2 \times y + 1]$ respectively. (Here the suffix "*ho*" corresponds to "associated string of holes", because the cycles of that string correspond to holes of the figure.)

One should note that with these two newly defined arrays, the association of a string to a chain extremity becomes indirect for the index of *beho* and *enho* is of type $max2$, which is the type used to number runs. Given a chain beginning $2 \times y + 1$ (where $y \in max2$), it is the right edge of the run numbered y, and y is the index used (in *beho* and *enho*) to access the string associated with the edge-element numbered $2 \times y + 1$. It looks as if the string was in fact associated to the run y instead of the chain extremity $2 \times y + 1$. In the following section, when discussing procedures working on chains and strings in *restricted surrounding*, we will indeed consider that strings are attached to runs and not to chain extremities. It will be convenient to visualize—from a purely topological viewpoint—the cycles of such strings as the outer cycles of small holes inside the corresponding runs.

To sum up, we have defined the following new data structures:

- *chex*, *sm*, *becs*, and *encs* in *full surrounding*,
- *chex*, *beho*, and *enho* in *restricted surrounding*.

There remains to be shown that they convey the information required by the real-time construction of cycles and edge-strings inside the procedure *allocate(u)*. This is the subject matter of the following subsection

5.4.4 Implementation

Recall from Subsection 4.5.1, that we decomposed the processing into three basic steps, *i.e.*, **initialization**, **processonrow**, and **transitiontothenextrow**. Let us first briefly make the case of both the former and the latter.

In the initialization stage and in *full surrounding*, the arrays *chex*[*v*], *sm*[*v*], *becs*[*v*], and *encs*[*v*] are set to -1, 0, *NIL*, and *NIL* respectively for $v = 0, \ldots, 4 \times maxnbr - 1$. In *restricted surrounding*, the arrays *beho*[*v*] and *enho*[*v*] are both set to *NIL* for $v = 0, \ldots, 2 \times maxnbr - 1$.

Recall also that the stage **transitiontothenextrow** consisted essentially of the assignments **precrow:=thisrow**, and **thisrow:=emptyrow**. Simultaneously, arrays *chex*, *sm*, *becs*(*ho*) and *encs*(*ho*) must be updated. In *full surrounding*, this is done by the following piece of code:

```
FOR x:=0 TO xn-1 DO                    {precrow:=thisrow}
  FOR a:=0 TO 1 DO
    BEGIN
    IF chex[4*x+a]=-1
      THEN chex[4*x+2+a]:=-1
      ELSE chex[4*x+2+a]:=chex[4*x+a]+1;
    sm[4*x+2+a]:=sm[4*x+a];
    becs[4*x+2+a]:=becs[4*x+a];
    encs[4*x+2+a]:=encs[4*x+a]
    END{a};

FOR x:=0 TO xm-1 DO                    {thisrow:=emptyrow}
    FOR a:=0 TO 1 DO
       BEGIN
       becs[4*x+a]:=NIL;
       encs[4*x+a]:=NIL
       END;
```

Updating the arrays *chex*, *sm*, *becs*, and *encs* at the stage "transition to the next row"

where **xn** $= max(precrow.nbr, thisrow.nbr)$ and **xm** $= thisrow.nbr$. With this example in mind, the case of *restricted surrounding* is self-explanatory.

Before we embark in a discussion of the construction of cycles and edge-strings, it will prove convenient to do justice to a minor technical detail which might appear to the alert reader as a slight inconsistency in the code formulation.

Let us examine the situation arising when the scan is at the end of $run[i, u]$, and both $run[i, u]$ and $run[i, u+1]$ are adjacent to a common run on row $i-1$. This situation is reflected by run parameters satisfying the conditions

$$u < nrisu[i-1, u'] - 1,$$

where

$$u' = nripr[i, u] - 1. \tag{5.26}$$

In this case, the portion of the image which has been scanned does not contain $led[i, u+1]$, which, in fact, follows $red[i, u]$, (see Table 5.2, case $D.3°$). So, $red[i, u]$ currently appears to be the end of a chain, which it is not. For this reason, we make the decision that the chain containing $red[i, u]$ ends in $led[i, u+1]$, and these two edge-elements make up a new chain. Now, the apparent inconsistency arises from the fact that, in the procedure *newobject*, adjacency relations between $run[i, u]$ and $run[i, u+1]$ are constructed in $allocate(u+1)$, while the chain link between $red[i, u]$ and $led[i, u+1]$ must be constructed in $allocate(u)$.

In the procedure *newobject*, equations (5.26) apply when the condition

```
IF (xrec.objty=1) OR ( (xrec.objty=0) AND (xrec.rri=u) )
```

is negated. Thus, after the "IF", we get an "ELSE" statement containing no operation on *objrec* records, but establishing the chain link between $red[i, u]$ and $led[i, u+1]$.

With this technical detail settled, let us return to the main stream of our preoccupations. The reader may find, in Appendix D, the code of six new procedures, nested in *allocate(u)*, namely: *concatenate, enclose, extendchain, newchain, mergechain,* and *closechain.* These procedures take two distinct forms according to whether we choose *restricted* or *full surrounding.* The first two ones are technical procedures used in *mergechain* and *closechain.* They operate on strings. The last four ones operate essentially on cycles. Hereafter, we shall examine all of them in some detail.

In *concatenate*, one starts from the situation where strings "S" and "T" are attached to chain extremities v and w respectively. When the chain ending in v is merged with that ending in w with v (only) loosing its former status of being a chain extremity, the two strings are concatenated into "ST" which is attached to w, with no string being attached to v any longer.

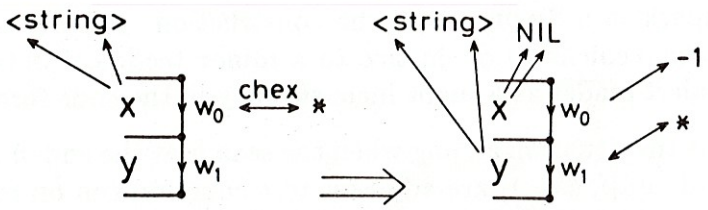


Figure 5.9. Procedure *extendchain* in *restricted surrounding.*

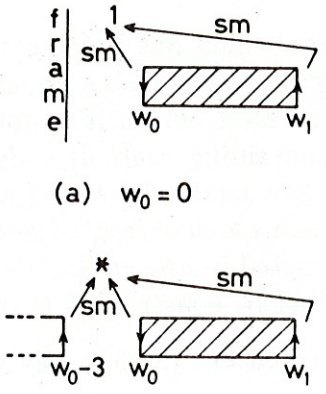


Figure 5.10. $sm[w_0]$ in procedure *newchain* in *full surrounding.*

The procedure *enclose* is activated at the time a chain is closed into a cycle. Its effect is to enclose a string "S" between the two occurrences (in the edge-string) of the newly formed cycle addressed by q : *cypoin*.

The next four procedures operate on chains and cycles, in particular, they put into effect, the edge-following operations described by Table 5.2. Hereafter, the references to Table 5.2 should not be taken too litterally, but rather as suggestive analogies.

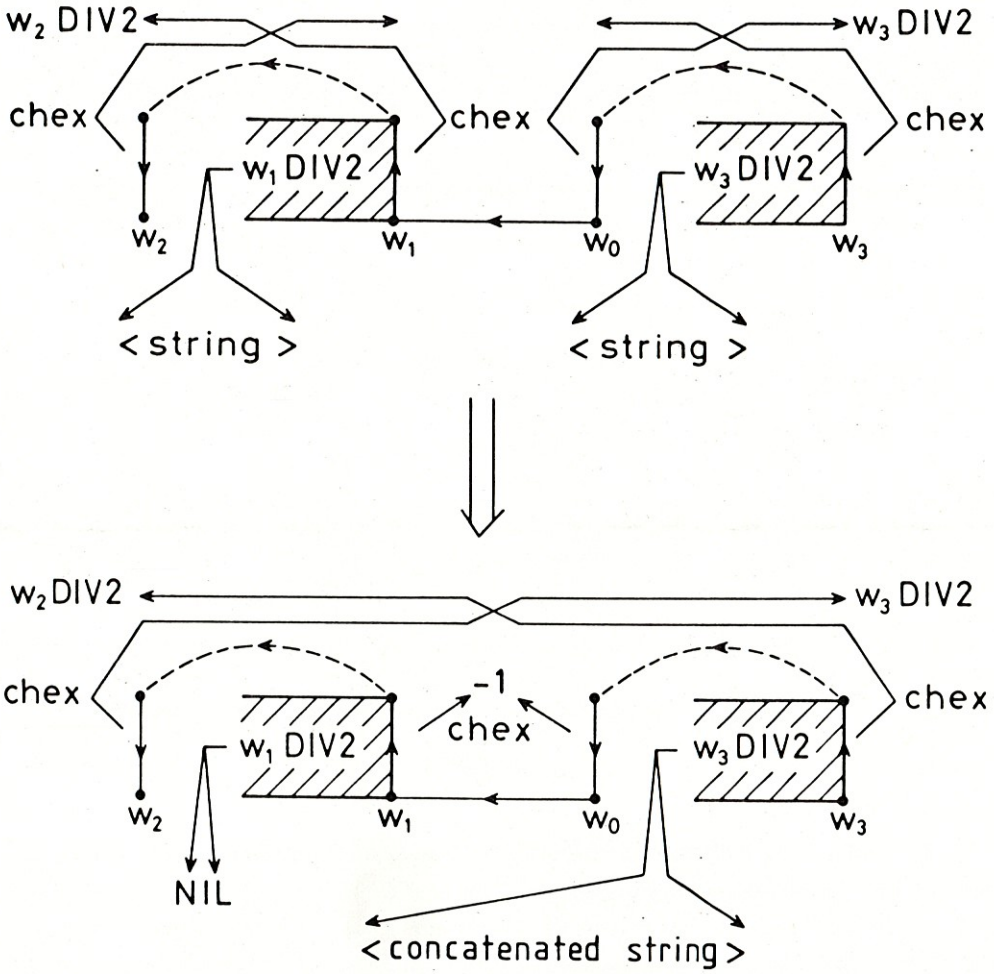


Figure 5.11. Procedure *mergechain* in *restricted surrounding.*

The procedure *extendchain* handles the cases *A*.2° and *D*.2° of Table 5.2: A chain is extended; one of its extremities is replaced by a new extremity. More precisely, let $w_0, w_1 \in max4$, such that $(w_0 \text{MOD} 4, w_1 \text{MOD} 4) = (2,0)$ or $(3,1)$ and $chex[w_0] \neq -1$. In plain words, w_0 and w_1 are edge-elements on the same side of their respective runs, and w_0 was a chain extremity. Let w_2 denote the opposite extremity of the chain containing w_0. If that chain is extended by w_1 on the side of w_0, then w_0 is no more a chain extremity, while w_1 becomes one. Moreover, the value of $chex[w_2]$ becomes $\lfloor w_1/2 \rfloor$ instead of $\lfloor w_0/2 \rfloor$. Finally, the string attached to w_0 must be concatenated with the one attached to w_1. This procedure is illustrated in Figure 5.9 in the case of restricted surrounding.

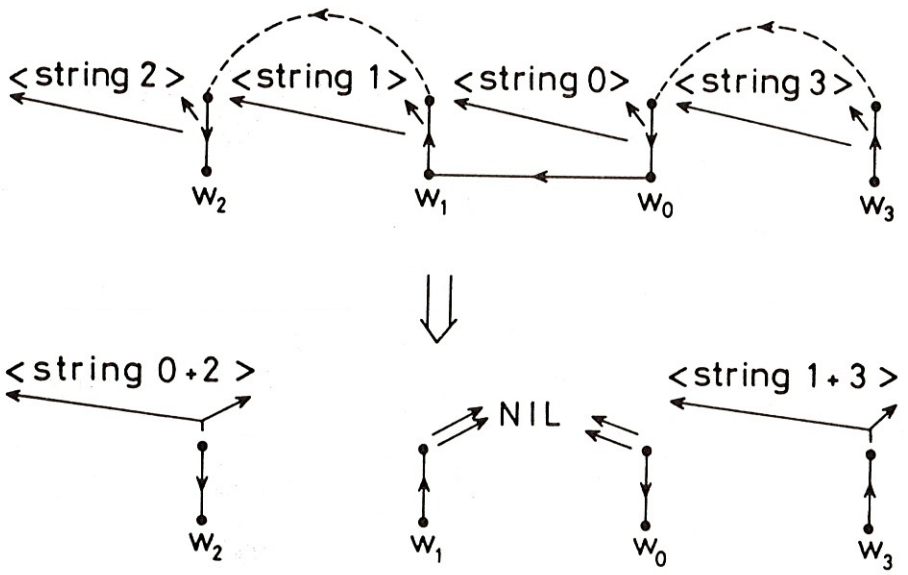
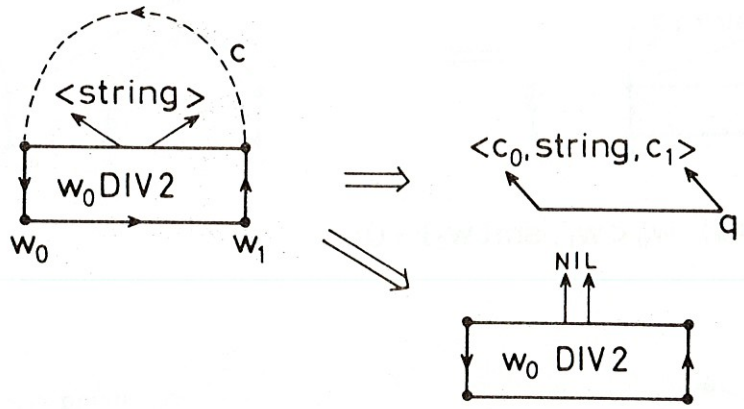


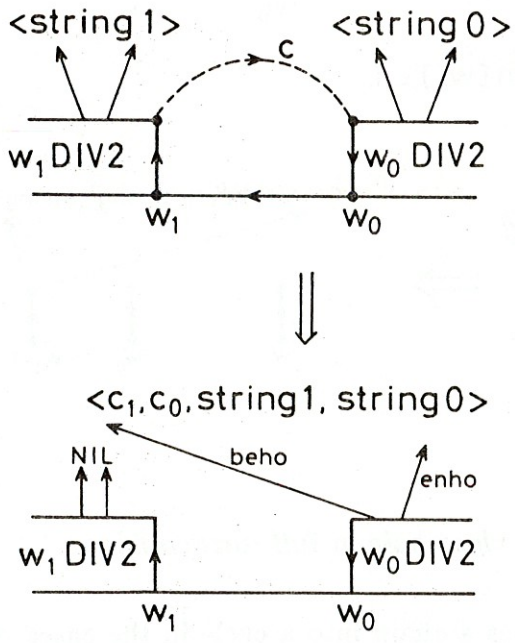
Figure 5.12. Concatenation of strings with *mergechain* in *full surrounding*.

The procedure *newchain* handles the cases $D.1°$ and $D.3°$ of Table 5.2: Given two newly considered edge-elements w_0 and w_1 satisfying the condition $(w_0\text{MOD}4, w_1\text{MOD}4) = (0, 1)$, they form together a new chain. Operations performed by *newchain* are rather straightforward except, perhaps, for the assignments of maximality numbers in full surrounding. Therefore, these are illustrated by Figure 5.10.

The procedure *mergechain* merges two distinct chains which become connected into a single chain. This may occurs with cases $A.1°$ and $A.3°$ of Table 5.2, and leads, inevitably, to changes in the array *chex*. Moreover, strings attached to former chain extremities must be concatenated with the strings attached to the remaining chain extremities. Note that the procedure uses as parameters the two disappearing chain extremities w_0 and w_1, where $(w_0\text{MOD}4, w_1\text{MOD}4) = (0, 1)$ or $(2, 3)$. The procedure *mergechain* is illustrated in Figure 5.11 in the case of restricted surrounding. Its effects on the arrays *becs* and *encs* in full surrounding are shown in Figure 5.12.

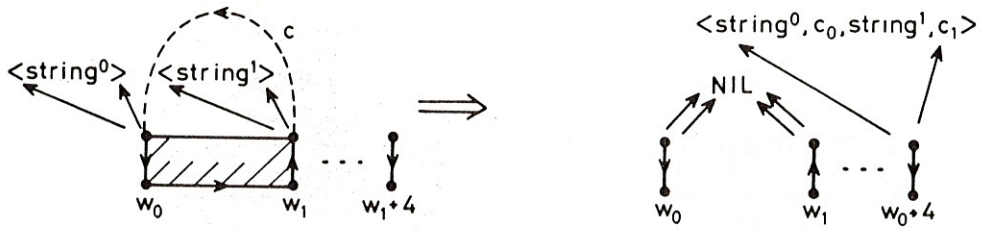


(a) $w_0 < w_1$

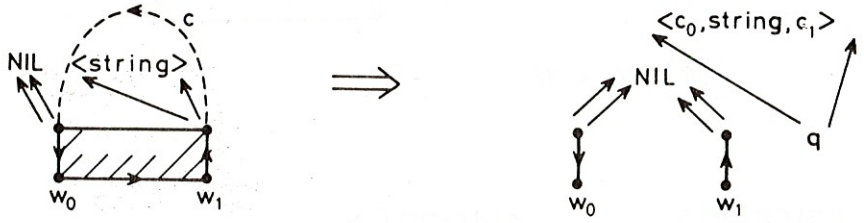


(b) $w_0 > w_1$

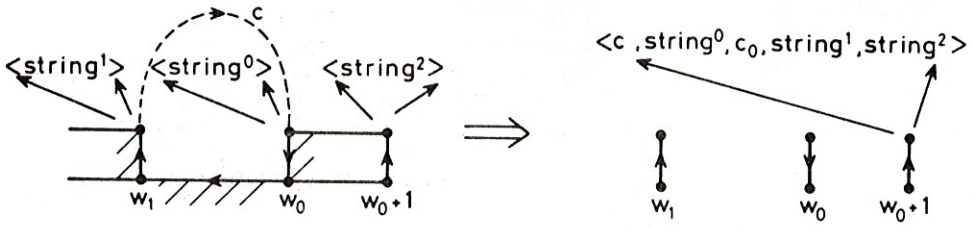
Figure 5.13. Procedure *closechain* in *restricted surrounding.*



(a) $w_0 < w_1$, $sm[w_0] = 0$.



(b) $w_0 < w_1$, $sm[w_0] = 1$.



(c) $w_1 < w_0$.

Figure 5.14. Procedure *closechain* in *full surrounding.*

The procedure *closechain* closes a chain into a cycle in the cases *A*.1° or *A*.3° of Table 5.2. Let the chain have beginning w_1 and end w_0 with (w_0MOD4, w_1MOD4) = (0, 1) or (2, 3). Then, the chain is closed into a cycle c with $c.access = z$ for a given pointer z. The procedure *closechain* is illustrated in restricted and full surrounding in Figures 5.13 and 5.14 respectively.

Next, let us see where these procedures can be called inside *allocate(u)*.

Recall first that the implementation of *restricted* and *full adjacency* in Section 5.2.4 induced some changes in the procedures *newobject* and *conbelow* only. Here again, it is in these two procedures only, and in the body of *allocate(u)* that the calls will be made. Thus, the eight procedures *endof, thisrowobjty0, thisrowobjty1, blockenlarge, continuationenlarge, newhinge, newblock, newcontinuation* remain unchanged. As in Section 5.2.4, we will have to refer to the detailed code as it appears in Appendices C and D.

(*i*) In *conbelow*, to the compound statement following "IF $consu = 0$ THEN" we add the following calls:

$$\begin{aligned} &\texttt{IF } chex[4 \times u] = 2 \times u \\ &\quad \texttt{THEN } closechain(4 \times u, 4 \times u + 1, z) \\ &\quad \texttt{ELSE } mergechain(4 \times u, 4 \times u + 1) \end{aligned} \tag{5.27}$$

In other words, when $consu[i, u] = 0$, we close the chain ending in $led[i, u]$ if $red[i, u]$ is the beginning of that chain, otherwise, we merge the two distinct chains containing these two edge-elements.

(*ii*) In *newobject*, we make several calls:

(*a*) To the compound statement following "IF $conpr = 0$ THEN" we add the call:

$$newchain(4 \times u, 4 \times u + 1) \tag{5.28}$$

In other words, if $conpr[i, u] = 0$ then, readily, $run[i, u]$ is the topmost run of a new object and $led[i, u]$ and $red[i, u]$ make up a new chain.

(*b*) In the "IF $x = 0$ THEN" part, we add to the compound statement following the "ELSE" of the condition "IF $u > 0$ AND $lefpr < pnp$" the call:

$$extendchain(4 \times lefpr + 2, 4 \times u) \tag{5.29}$$

In other words, if $consu[i, u] > 0$ and $nripr[i, u-1] \leq lefpr[i, u]$, then $led[i, u]$ extends the chain ending in $led[i-1, lefpr[i, u]]$.

(*c*) In what follows "IF $x = conpr - 1$ THEN" we have the condition

```
IF (xrec.objty=1) OR ( (xrec.objty=0) AND (xrec.rri=u) )
```

which means that $run[i, u]$ and $run[i, u+1]$ are not adjacent to a common run above them. In the compound statement following the "**THEN**" of that condition, we add the call:

$$extendchain(4 \times nripr - 1, 4 \times u + 1) \tag{5.30}$$

In other words, if for $v = nripr[i, u] - 1$, $u = nrisu[i, v] - 1$, then $red[i, u]$ extends the chain beginning in $red[i-1, v]$.

(*d*) In the compound statement following the "**ELSE**" of the condition "**IF** $x = 0$" we make the assignment

$$v := 4 \times (lefpr[i, u] + x) + 2$$

and the call

$$\begin{aligned} &\textbf{IF } chex[v] = v \text{ DIV } 2 - 2 \\ &\quad \textbf{THEN } closechain(v, v-3, s) \\ &\quad \textbf{ELSE } mergechain(v, v-3) \end{aligned} \tag{5.31}$$

where v DIV 2 denotes $\lfloor v/2 \rfloor$. Recall that s is of type *link*, and points to the object containing $run[i-1, lefpr[i, u] + x]$. The call closes the chain containing $led[i-1, lefpr[i, u] + x]$ and $red[i-1, lefpr[i, u] + x - 1]$ if they are on the same chain, otherwise it merges the two chains containing them.

(*e*) To the condition of (*c*), we add an "**ELSE**" statement containing only the call

$$newchain(4 \times u + 4, 4 \times u + 1) \tag{5.32}$$

In other words, if $run[i, u]$ and $run[i, u+1]$ are both adjacent to $run[i-1, nripr[i, u] - 1]$, then we create a new chain consisting of $red[i, u]$ and $led[i, u+1]$ in accordance with our discussion at the beginning of this section.

(*iii*) In the body of *allocate*, we add to the two compound statements containing respectively the calls of *blockenlarge* and *continuationenlarge* the two calls:

$$extendchain(4 \times lefpr + 2, 4 \times u)$$

and

$$extendchain(4 \times lefpr + 3, 4 \times u + 1). \tag{5.33}$$

Indeed, when a block or block-continuation is increased by one run, we must extend the chains on both sides. This completes the list of modifications required in the three procedures *conbelow, newobject* and *allocate.*

Among the six new procedures introduced in this section, namely, *concatenate, enclose, extendchain, newchain, mergechain,* and *closechain,* the last one is playing the prominent role. Indeed, it is in it that we construct the cycles and we can insert the output command. Specifically, at the end of the compound statement following the "IF $w_0 < w_1$ THEN"—which corresponds to the case where the newly constructed cycle is the outer cycle of a connected component of F—we can call the output procedure (see next chapter) which does the following:

- In *full surrounding,* it takes that connected component out of the main memory, and transfers it to the output buffer. If, in addition, $sm[w_0] = 1$, then it also transfers the string enclosed by that cycle to the output buffer.
- In *restricted surrounding,* it transfers that connected component and the associated simplified edge-string from the main memory to the output buffer.

The output procedures are discussed at some lenght in the following chapter.

§5.5 Comments

As in previous chapters, we wish to conclude with some considerations of the time- and space-complexities associated with the various extensions of procedure *allocate* that were presented above.

As far as time is concerned, we explained in Chapter 4 that, by calling procedure *allocate* at the end of the scan of each run, the processing was in real-time at the run level, and the time involved was proportional to the size of the run under consideration. The additions described in this chapter essentially consist in the processing described by Eqs. (5.8)–(5.13) and (5.27)–(5.33). It should be evident that these additions do not impair in any way the real-time property of the process.

The memory requirements have been increased in two ways. On the first hand, we have added the arrays *chex, becs(ho), encs(ho),* and *sm* whose size is linear in *maxnbr* (as was the case for *precrow* and *thisrow* in Chapter 4). This space requirement is, at worst, linear in N.

On the other hand, we have slightly expanded the *objrec* records and we have created the *cyrec* records corresponding to the connected components of F and their holes. Clearly, *objrec* and *cyrec* records are used to accumulate the information which we have chosen to make available at the output. The number of such records existing at any time in working memory depends very much on the kind of picture being processed. In any event, dynamic memory management permits to keep that number to a minimum.

REFERENCES

[1] O.P. Buneman, "A grammar for the topological analysis of plane figures," in *Machine Intelligence 5,* B. Meltzer and D. Michie Eds, Edinburgh University Press, 1969, pp. 383–393.

[2] R. Cederberg, *On the Coding, Processing, and Display of Binary Images,* Ph.D. Dissertation No. 57, Linköping Univ., Dept. Electr. Engrg., Linköping, Sweden, 1980.

[3] B. Kruse, "A fast algorithm for segmentation of connected components in binary images," *Proc. first Scandinavian Conf. Image Analysis,* Lund, Sweden: Studentlitteratur, 1980, pp. 57–63.

[4] C. Ronse, *Digital Processing of Binary Images on a Square Grid,* Philips Res. Rept. [R.454], June 1981.

[5] A. Rosenfeld, "Adjacency in digital pictures," *Information and Control,* vol. 26, pp. 24–33, 1974.