An Improved Algorithm for Labeling Connected Components in a Binary Image

Xue Dong Yang*

TR 89-981
March 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Work on this paper was supported in part by ONR grant N00014-87-K-0129, NSF CER grant DCR 83-20085, NSF grant subcontract CMU-406349-55586, grants from DEC and IBM. Also supported by ONR grant N00014-88-K-0591 NSF grant DMC-86-17355 and ONR grant N00014-86-K-0281.

An Improved Algorithm for Labeling Connected Components in a Binary Image

Xue Dong yang[1]
Cornell University
Computer Science Department
Ithaca, NY 14853

March 22, 1989

[1]Work on this paper was partly done in the Robotics Research Laboratory of Courant Institute, New York University and has been supported by ONR Grant N00014-87-K-0129, NSF CER Grant DCR-83-20085, NSF Grant subcontract CMU-406349-55586, and by grants from the DEC and the IBM; and the work is finished at the Computer Science Department of Cornell University under the supports from DARPA ONR grant N0014-88-K-0591, NSF grant DMC-86-17355 and ONR grant N00014-86-K-0281.

Abstract

In this note, we present an improved algorithm to Schwartz, Sharir and Siegel's algorithm [8] for labeling the connected components of a binary image. Our algorithm uses the same bracket marking mechanism as is used in the original algorithm to associate equivalent groups. The main improvement of our algorithm is that it reduces the three scans on each line required by the original algorithm in its first pass into only one scan by using a recursive group-boundary dynamic tracking technique, while maintaining the computation on each pixel during scan still a constant time. This algorithm is fast enough to handle images in real time and simple enough to allow an easy and very economical hardware implementation.

List of Symbols

m	the number of rows of an image
n	the number of columns of an image
R_p	the row p of an image
I_p	lower semi-image from row p to row m
r	a run in a row defined as a sequence of 1-pixels bounded on both side by 0-pixel
G_p	partition of a set of runs in R_p

1 Introduction

The labeling of connected components of a binary image is a fundamental problem in image analysis. An early method was developed by Rosenfeld and Pfaltz [7] in 1966; it uses a pair of arrays, one containing the current region label and the other containing its smallest equivalent label. This algorithm processes an image from top to bottom to compute label equivalences, storing the result in the arrays. A second pass reassigns each label to its smallest equivalent label. Lumia, Shapiro and Zuniga [3] improved this method by using a short equivalence table, which needs to cover only one line. Schwartz, Sharir and Siegel [8] presented an algorithm which uses **bracket marking** to associate equivalent groups. This method enables one to compute the component numbers for each pixel on the fly, by using an relative small auxiliary **bracket table**. More interestingly, this algorithm uses mainly pushdown-stack data structures which allows simple high-speed hardware implementation. In addition to the above mentioned sequential algorithms, there are parallel algorithms, for example, an logarithmic-time connected components algorithm for massively parallel computing system (e.g. one processor per pixel) connected in shuffle-exchange or other similar pattern by Shiloach and Vishkin [9].

In this note we present an improved algorithm to the Schwartz, Sharir, and Siegel's algorithm [8]. Like the original algorithm, we make two passes over a binary image, with the first pass sweeping row by row from the bottom to top of the image and the second pass in the opposite direction. However, our improved algorithm makes only one scan on each line in the first pass, while the original algorithm needs three scan on each line in the first pass. While not as fast as the logarithmic-time parallel algorithm [9], the algorithm to be presented is fast enough to handle images in real time and simple enough to allow an easy and very economical hardware implementation.

The algorithm described in this note has been implemented in hardware [10]. A prototype

Connected Components Board has been designed and physically implemented by the author in the Robotics Research Laboratory of New York University. It did not involve any specially designed VLSI chips and can compute a 512 by 512 binary image in about 300 ms. A real-time version of the algorithm in VLSI, which pipelines the two passes of the algorithm, has been proposed in [10].

In the following, we first review the definitions and details of Schwartz, Sharir, and Siegel's algorithm [8]. Then, we describe our improved algorithm.

2 The Original Algorithm

Assume that a binary image has m rows and n columns. Some key definitions of the original algorithm are reviewed as below.

Definition 1: *Let R_p be an image row, $1 \leq p \leq m$.*

(a) A **run** *in R_p is a sequence of 1-pixels (i.e. pixels with gray value 1) of R_p bounded on both sides by 0-pixel (For simplicity, we add two additional 0-pixels to the left and right-end of each row of the image, respectively).*

(b) The **lower semi-image** *I_p consists of the union of all rows R_j, $p \leq j \leq m$.*

(c) G_p is defined to be the partition of the set of runs in R_p, for which two runs are in the same **partition group** *$g \in G_p$, iff they belong to the same connected component of the lower semi-image I_p.*

The original algorithm consists of two passes. *Pass 1* sweeps through the rows from bottom to top, during which the groups in G_{p-1} are calculated inductively from the knowledge of R_p by a simple updating rule. *Pass 2* sweeps from top to bottom to assign component numbers to each pixel and outputs this symbolic image.

In each row, runs belonging to the same group are associated by a simple mechanism, called

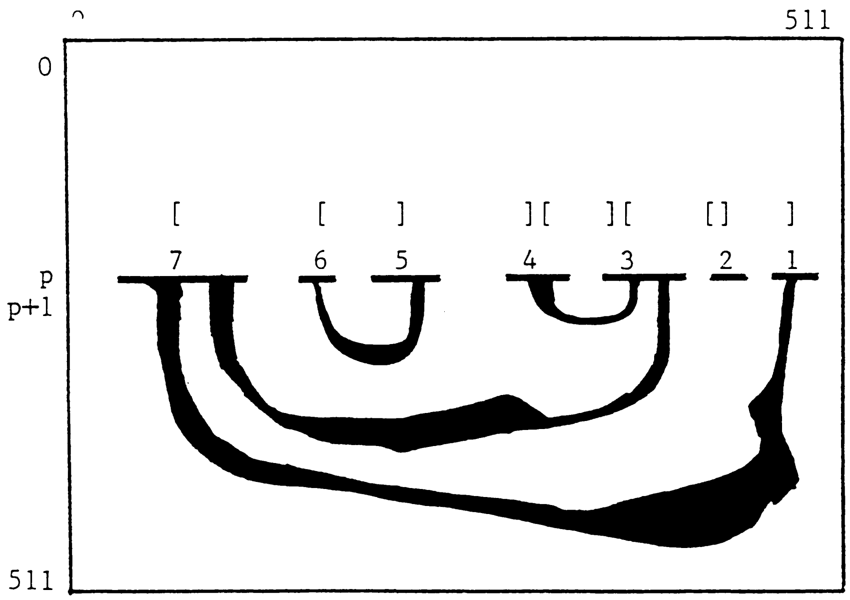

Figure 1: An example of bracket marking assignment

bracket marking, as follows:

Definition 2: *We define four symbols '[', ']', '[]', and '][*'.

(a) If r is the leftmost (resp. rightmost) run in its group g, it is assigned marking [(resp.]).

(b) If r is both leftmost and rightmost run in g, i.e. g consists of a single run r, then r is assigned the marking [].

(c) If r lies between the leftmost and rightmost run in its group, it is assigned the marking][.

An example is given in Fig 1, where a bracket marking is shown for seven runs in R_p numbered from right to left. These seven runs are divided into three groups – (7,4,3,1), (6,5), and (2).

The following lemmas have been proved in [8]:

Lemma 1: *(a) the bracket sequence that the proceeding definitions associate with the row R_p is properly nested, i.e. each right bracket in it is matched (by the well-known stacking algorithm) to an associated left bracket and vice versa.*

(b) The groups into which we have divided the set of all runs in R_p can be reconstructed from the bracket sequence associated with R_p by applying the following rule: put all runs whose

associated brackets match into one group. (Note that according to this rule runs with the '][' marking will link certain runs to their left and certain runs on their right into a single group.)

Lemma 2: *Let g and g' be distinct groups of runs in G_p. Then if there are runs x_1, x_2 in g, which are to the left and to the right, respectively, of some run x' in g', it follows that all the runs in g' lie between x_1 and x_2.*

The goal of *pass 1* is to calculate bracket marking from bottom to top row by row. For this we want a rule telling us how to calculate the groups (or, equivalently, the bracket marking) for the row R_{p-1} given the same information for R_p. Our aim is just to determine which runs of R_{p-1} have other runs in the same group which lie to their left (resp. right). Note that two runs, say r_i and r_j for some i and j, of R_{p-1} belong to a same group if:

(a) both of them overlap with some run of a same group in R_p; or

(b) r_i overlaps with a run of a group in R_p, r_j overlaps with a run of a different group in R_p, and these two groups in R_p are then merged together by some run other than r_i and r_j in R_{p-1}.

The original algorithm makes four scans on each image row, two left-to-right, the others right-to-left. The first two of these scans calculates what is called **extended group**:

Definition 3: *Two runs in R_p belong to the same* **extended group** *if they are members of the same connected component of the lower semi-image I_{p-1}.*

Note that every extended group g' of R_p is a union of one or more groups g of R_p which are merged together by some runs in R_{p-1}. The merging of two groups can be in following three basic ways:

Case (a)

In case (a) (Fig. 2.a), a run of R_{p-1} overlaps with both the leftmost run of group g_i and the rightmost run of group g_j of R_p. To introduce an equivalence between two immediately adjacent brackets ']', '[', we can simply change them both to ']['.

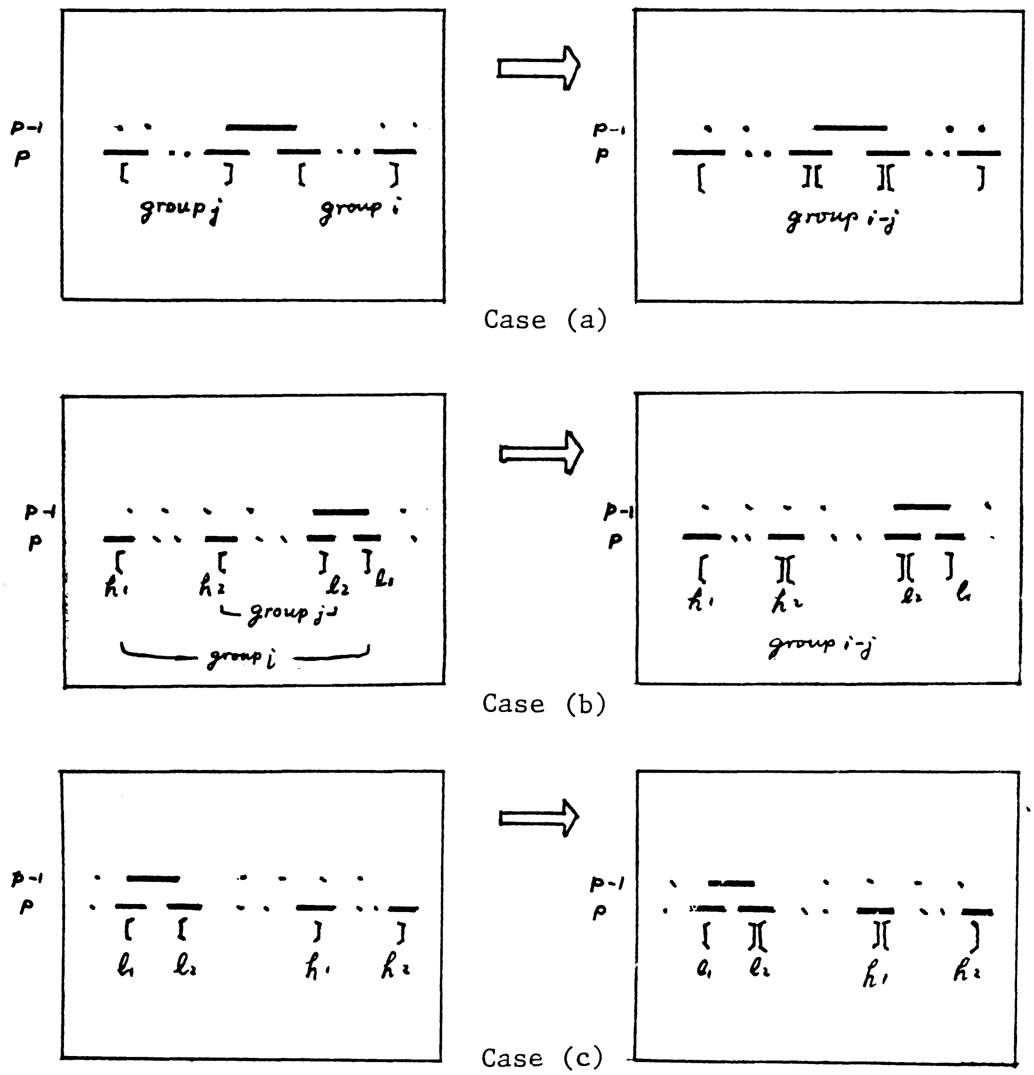

Figure 2: Three basic ways of merging two groups in calculating extended group.

Case (b)

In case (b) (Fig. 2.b), the equivalence between group g_i and group g_j can be introduced during the scan from right to left as following. We can change l_2 from ']' to '][' and leave l_1 unchanged; then we must locate the extreme left-hand bracket h_2 of group g_i (in R_p) whose rightmost bracket is l_2, and change h_2 from '[' to ']['.

Case (c)

The case (c) is symmetric to the case (b). Hence it can be handled in a similar manner of the case (b) in an opposite scan from left to right.

It follows that the extended groups of R_p can be obtained by making the bracket modifications described. This modification can be made in two successive passes (over R_p and R_{p-1} together), as follows: scan the pixels of R_p and R_{p-1} simultaneously, from right to left, holding unmatched brackets on a stack. Whenever runs of R_p described by successive brackets ']', ']' (call them $l_2$and l_1 as before) are found to be in contact with an unbroken run in R_{p-1}, change l_2 to ']*[', where '*' is an additional 1-bit mark which indicates that the bracket representing the left end of the group whose right end is l_2 must be changed to '][' when encountered during the stacking/unstacking process. Treat successive pairs of brackets '[', '[' symmetrically during an immediately following left-to-right scan.

Once these bracket modifications have been performed, two more relatively straightforward passes, over R_p and R_{p-1} simultaneously, suffice to construct a bracket marking describing the groups of R_{p-1}.

The third scan (left-to-right) identifies all those runs r in R_{p-1} which belong to groups G containing runs r' lying to the left of r. We still scan the rows R_{p-1} and R_p together, and simultaneously scan the modified bracket marking in R_p. An auxiliary stack S is used to store left brackets discovered during the scan of R_p that have not yet been matched. If a run r in R_p

is currently being scanned, then the bracket on top of S will describe the group containing r. Stacked brackets have two mark fields: **grouphit** and **old**. The bracket on top of S will have its **grouphit** mark set to 1 whenever, during the scan of R_{p-1} a pixel in R_p is discovered to be adjacent to a pixel in R_{p-1}. This records the fact that some run in the group represented by the top bracket (and all matching brackets) is known to be adjacent to some run of R_{p-1}. The **old** mark distinguishes between the case in which a group g of R_p represented by a stacked bracket with **grouphit** $= 1$ only has pixels adjacent to the run in R_{p-1} that is currently being scanned (in which case **old** $= 0$), from the contrary case in which g is adjacent to a run in R_{p-1} that has already been scanned (in which case **old** $= 1$).

The start of each run r of R_p pushes an associated left bracket on S if the marking of r is either [or [], and the end of r pops the top bracket of S if r is marked either] or []. The marking][is handled most efficiently by regarding it as a 'no-op' which simply continues the bracket currently on the stack. Whenever two adjacent white bits, belonging to runs $r' \in R_{p-1}$ and $r \in R_p$ respectively are seen, we check whether the top bracket in the stack is **grouphit** and **old**. If this is the case, r' must be connected to some run in R_{p-1}, and that run lies to the left of r'. If not, the bracket's **old** mark must be zero, since runs in r's group lying to the left of r do not contact R_{p-1}; but in this case the **grouphit** mark of this same bracket will already be set.

The subsequent right-to-left scan performs exactly the mirror image of these actions, i.e. determines what runs r of R_{p-1} are part of a group extending to their right. This gives us the bracket marking associated with R_{p-1}.

Since the second scan produces each modified bracket just when this is needed by the third left-to-right scan, these two left-to-right scans can be combined into one. Thus three scans over each successive pair of rows suffice to generate the bracket markings in R_{p-1}.

We then perform a top-to-bottom pass which completes the assignment of connected component numbers. To process the groups g of G_p after R_{p-1} has been processed, we simply apply the following rule: if any run in g touches any run r in R_{p-1}, assign each pixel of g the same component number as that assigned to r. Otherwise g represents a new component; assign a new component number to its pixels. To do this, we perform a simultaneous left-to-right pass over R_p and R_{p-1} during which the parenthesis marking of R_p drives the stacking/unstacking procedure previously described; during this process each stacked bracket must be marked either with a zero (indicating no contact yet), or with a nonzero integer defining a component number. While this is done, a queue giving the component number for each group of R_{p-1} must be available. Component numbers can be stored at the end of the final run of the corresponding group of R_p.

3 An Improved Algorithm

Our improved algorithm makes also two passes over an image and uses the same bracket marking mechanism as that of the original algorithm. The major difference of our algorithm from the original algorithm is that our algorithm reduces the three scans on each row required by the original algorithm in its first pass into only one scan while maintaining the computation on each pixel during the scan still in a constant time. The main idea of this improvement is the following. Instead of first calculating the **extended groups** of R_p before actually computing the bracket marking of R_{p-1} as is done in the original algorithm, we now directly compute the bracket marking of R_{p-1} in one scan by using a recursive group-boundary dynamic tracking technique.

Assume that we scan a row from right to left. The connectivity of runs in row R_{p-1} through links in the semi-image I_p can be classified into three basic cases as follows (refer to Fig. 3):

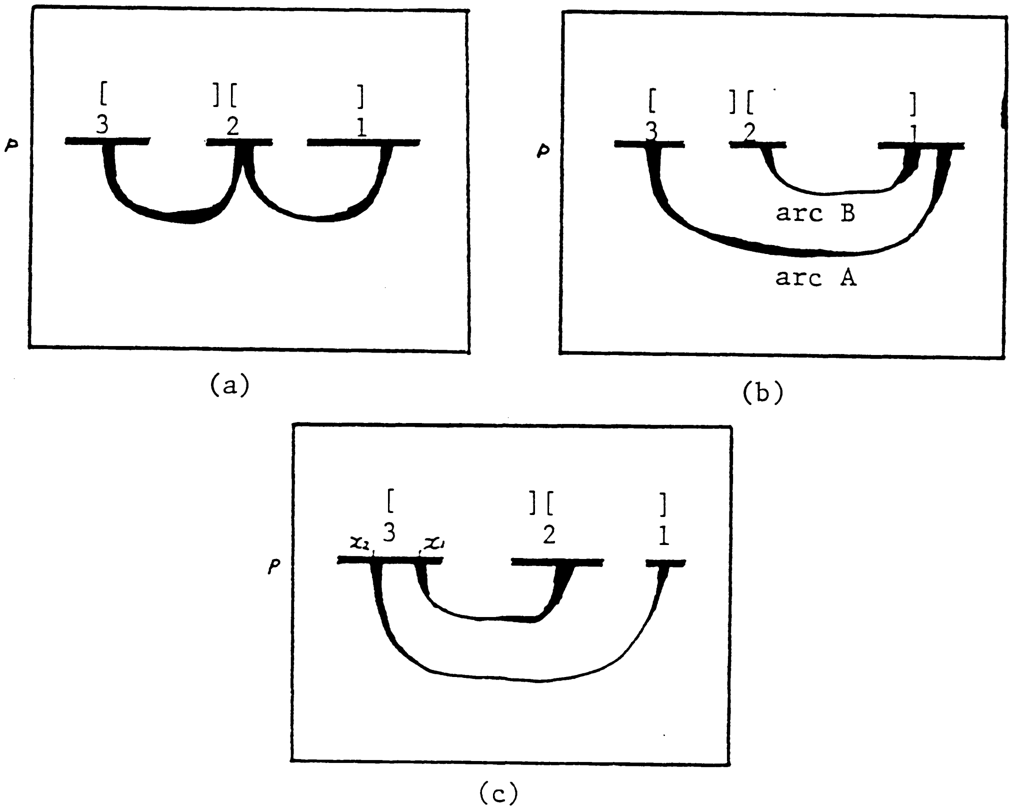

Figure 3: Basic cases of connectivity in semi-image I_p: (a) simple chain; (b) a cluster of links is bound at right end; (c) a cluster links is bound at left end.

(a) Several runs may be linked by a simple chain;

(b) A cluster of links (or arcs) may be bound together on its right end by a single run in row R_p;

(c) A cluster of links (or arcs) may be tied together on its left end by a single run in row R_p.

The general connectivity can be decomposed into these three elementary cases by recursively considering a subset of consecutive runs in a group as a virtual "single run". For example in Fig. 1, if we consider runs 4 and 3 as a virtual "single run" denoted by 4-3, the connectivity between 7, 4-3, and 1 falls into the elementary case (c). Our algorithm handles the connectivity problem from inner level to outer level recursively. Thus, at any time, we only have to deal with one of the three elementary cases defined above; however a run could be a simple run or a virtual "single run".

We now use two stacks, called *stack1* and *stack2*. *Stack1* plays a similar role as in the original algorithm to trace the bracket marking already computed for R_p, while *stack2* is used to dynamically trace group boundaries of R_{p-1} during the scan. Assume we number each run in row R_{p-1} from right to left starting from 1, as an example shown in Fig. 1. We still scan R_{p-1} and R_p together. At each stage of the scan , there is an entry in *stack2* for each non-ended group found so far. Each entry is a pair of integers representing the current right-most and left-most runs of the corresponding group. In the following we show how this group boundary information can be updated according to the new knowledge we gain about each group as the scan continues.

When a run, say r_j, of R_{p-1} starts a new group, an entry $[r_j,r_j]$ is pushed on *stack2*. The condition for starting a new group can be easily checked out locally during the scan. Specifically, when we encounter a new run in row R_{p-1}, we check if it makes contact with any run in row R_p. If it does not, we know that it is the start of a new group containing one isolated run only; Or it does make contact with some runs in R_p the right-most of which has a ']' marking, we assume temporarily that a new group starts. (We say 'temporarily' here since initially no link connects the current run in R_{p-1} to any run to its right on the same row. However this new run might link to some run to its right indirectly through several arcs. For example in Fig. 3(c), run 2 links indirectly to run 1 through two arcs.)

Now we describe how a group's boundaries are expended in each of three elementary cases and when a group is terminated.

Case (a)

In case (a) of Fig. 3, we have entry [1,1] on *stack2* after reaching run 1, which means the left and right boundaries of the current group are both equal to 1. When run 2 is encountered, we know that it links to its right to the current group represented by the top of *stack2* by checking the top of *stack1* and applying lemma 2. So the left bound expands to 2 after run 2, resulting in the top stack entry [1,1] being updated to [2,1]. Similarly, it is then updated to [3,1] when run 3 is reached. When a run of R_p with a '[' marking is scanned, the top entry of *stack1* is popped off and also if the group g of R_p represented by this popped entry contacts the current group g' of R_{p-1} represented by the top entry of *stack2*, then g' is terminated since there is no more path in I_p possible to link g' further to its left; hence, its entry on *stack2* is popped off.

Case (b)

The situation in case (b) of Fig. 3 is slightly more complicated. We have [1,1] on *stack2* after run 1. (Remember that we have two marked entries on *stack1* after run 1 in row R_p since it contacts two runs in row R_p both of which have ']' brackets.)

After run 2, the top entry on *stack2* becomes [2,1] and the top entry on *stack1* was popped. At this point we do not know whether or not to pop the top entry on *stack2*. While it is possible, using only local information to determine the start of a new group, this is not the case for determining the end of the current group. The fact that a run has no more arcs linking it leftwards does not necessarily mean that the end of this group has been reached. (Note that run 2 is further linked to run 3 through an indirect path.)

To handle this situation, we introduce an auxiliary field in *stack1*'s entry to indicate whether the arc associated with a ']' marking in that entry is the outermost (or lowest) arc. For example, we say arc A is the lowest among the cluster of arcs bounded by run 1 in row R_p in (b) of Fig. 3. The auxiliary bit can be easily set as following. During an unbroken run in row R_{p-1}, if one or more entries have to be pushed onto *stack1*, the auxiliary field in the first pushed entry is set to TRUE and the auxiliary fields in all other entries are set to FALSE. If the auxiliary field in the entry just popped off *stack1* is FALSE, we know the arc (or link) just ended is not the outermost (lowest) one, i.e. there is an arc belonging to the same group enclosing this finished arc, so we don't pop the entry off *stack2* because we don't know if the current group in row R_{p-1}

is finished yet. After reaching run 3 in row R_p, the left bound of current group is expanded to 3, and the top entry on *stack2* is changed from [2,1] to [3,1]. When the entry on *stack1* is popped off, we see that the auxiliary field is TRUE. So, we can confidently pop the top entry on *stack2* off this time because we know that there is no further arc linking this group to any run to its left.

Notice that, since the introducing of this auxiliary field, the handling of case (a) should be modified to include a checking of this auxiliary field in determining the termination of the current group.

Case (c)

A situation symmetrical to case (b) occurs in case (c). After reaching run 1, we have one entry, [1,1], on *stack2*. Note that at the moment after reaching run 2, it is impossible to know that this run is actually linked to run 1 through an indirect path. So, we have a second entry, [2,2], on top of *stack2*. After point x_1 in run 3 is reached, the top entry of *stack2* is updated from [2,2] to [3,2]. When we reach point x_2 we find that two arcs are bound together. So, the two top entries on *stack2* are merged into one in such a way that the new left bound is the left bound of first entry on *stack2* and the new right bound is the right bound of the second entry of *stack2*, resulting a [3,1] on *stack2*. These operations can be performed in two steps: (1) pop *stack2*; (2) replace the left bound of top entry by the left bound of the entry just popped. In this example, we pop [3,2] off *stack1* first; then change the left field of top entry - [1,1] to 3 and get [3,1] as the new top on *stack1*.

The bracket marking can be effectively encoded by a two-bit binary number as shown in Table 1. The bracket information calculated during *pass 1* is stored in a memory of $m \times \frac{n}{2} \times 2$ bits, called the *bracket table*. This table is indexed by two row counters, *row_count1* and *row_count2*, and two column counters, *run_count1* and *run_count2*. *Row_count1* and *run_count1* are combined

Bracket Marking	Left Bit	Right Bit
[]	0	0
[	0	1
]	1	0
][	1	1

Table 1: Bracket Marking Encode Definitions

to access brackets for row R_{p-1}, and *row_count2* and *run_count2* for those in row R_p. Each entry in the table has two 1-bit fields, both of which are initialized to zero. *Run_counts* are set to zero at the beginning of a row scan and incremented by one every time a new run started.

Now we show how the bracket table can be updated efficiently as the group boundaries updated dynamically during a scan. There are only two types of group-expanding operations: (a) the left boundary of the current group expands to include the current scanned run; (b) the two top most groups on *stack2* are merged together. To do each of these operations in constant time, we must be able to directly index correct columns in a row of the bracket table so that we can set corresponding bits to 1's. The index of the current scanning run is provided by *run_count1* for row R_{p-1} and *run_count2* for row R_p. The boundaries of all non-ended groups are maintained in *stack2* in a properly nested order with the current group on the top. Thus, the updating of the bracket table's entry in every possible case can obviously be done in a constant time.

The second pass of our algorithm is same as that of the original algorithm. A full implementation of out algorithm written in C programming language can be found in [11].

4 Conclusion and Additional Remarks

In this note, we have presented an improved algorithm to Schwartz, Sharir and Siegel's algorithm [8] for labeling the connected components of a binary image. Our algorithm uses the same bracket marking mechanism as is used in the original algorithm to associate equivalent groups. The main improvement of our algorithm is that it reduces the three scans on each line required by the original algorithm in its first pass into only one scan by using a recursive group-boundary dynamic tracking technique, while maintaining the computation on each pixel during scan still a constant time. This algorithm is fast enough to handle images in real time and simple enough to allow an easy and very economical hardware implementation. In fact, a prototype connected components board has already been designed and implemented by the author [11].

When we want to compute a sequence of input images continuously, it is interesting to pipeline the two passes of the algorithm in order to get a sequence of continuous output symbolic images. I some applications, it will be useful to identify the k largest components and/or to calculate various additive geometric invariants of these components, e.g. their number of pixels, medians and second moments. These computations can be performed in a variety of ways. One simple method is to compute these values on the fly during the top-to-bottom (i.e. second) pass; or separate these computations into an individual stage and place it into the pipeline at the place after the second pass.

Acknowledgements

I had valuable discussions with Prof. Jacob Schwartz, Prof. Alan Siegel, Prof. Robert Hummel, Mr. Alan Kalvin and Mr. Eric Freudenthal.

References

[1] Ballard, D.H., Brown, J.E., *Computer Vision*, Prentice-Hall, Inc., NJ, 1982

[2] Dinstein, I., Yen, D., Flickner, M., *Handling Memory Overflow in Connected Component Labeling Applications*, IEEE Trans. on PAMI, Vol. PAMI-7, No. 1, Jan. 1985

[3] Lumia, R., Shapiro, L., Zuniga, O., *A New Connected Components Algorithm for Virtual Memory Computers*, Computer Vision, Graphics, and Image Processing 22, 287-300, 1983

[4] Rosenfeld, A., *Adjacency in Digital Pictures*, Information and Control, Vol. 26, No. 1, Sept. 1974

[5] Rosenfeld, A., Kak, A.C., *Digital Picture Processing, Vol. 1, 2*, Academic Press, 1982

[6] Rosenfeld, A., *Picture Processing: 1975*, Computer Graphics and Image Processing, 5, 215-237, 1976

[7] Rosenfeld, A., Pfaltz, J.L., *Sequential Operations in Digital Processing*, JACM, 13, 471-494, 1966

[8] Schwartz, J.T., Sharir, M., Siegel, A., *An Efficient Algorithm for Finding Connected Components in a Binary Image*, Technical Report No. 154, Courant Institute, NYU, 1985

[9] Shiloach, Y., Vishkin, U., *An O(log n) Parallel Connectivity Algorithm*, Journal of Algorithms 3, 57-67, 1982

[10] Yang, X.D., *Design of Fast Connected Components Hardware*, The Proceedings of IEEE Computer Vision and pattern Recognition, Ann Arbor, MI, June 5-9, 1988.

[11] Yang, X.D., *Design of Fast Connected Components Hardware*, (detialed version), Technical Report No. 353, Courant Institute, NYU, 1988