Fast Run-Based Connected Components Labeling for Bitonal Images

Wonsang Lee,[1] Stefano Allegretti,[2] Federico Bolelli,[2] and Costantino Grana[2]

[1] Department of Physics
Konkuk University, Korea
Email: attltb@gmail.com

[2] Dipartimento di Ingegneria "Enzo Ferrari"
Università degli Studi di Modena e Reggio Emilia, Modena, Italy
Email: {*name.surname*}@unimore.it

***Abstract*—Connected Components Labeling (CCL) is a fundamental task in binary image processing. Since its introduction in the sixties, several algorithmic strategies have been proposed to optimize its execution time. Most CCL algorithms in literature, including the current state-of-the-art, are designed to work on an input stored with 1-byte per pixel, even if the most memory-efficient format for a binary input only uses 1-bit per pixel. This paper deals with connected components labeling on 1-bit per pixel images, also known as 1bpp or bitonal images. An existing run-based CCL strategy is adapted to this input format, and optimized with *Find First Set* hardware operations and a smart management of provisional labels, giving birth to an efficient solution called Bit-Run Two Scan (BRTS). Then, BRTS is further optimized by merging pairs of consecutive lines through bitwise *OR*, and finding runs on this reduced data. This modification is the basis for another new algorithm on bitonal images, Bit-Merge-Run Scan (BMRS). When evaluated on a public benchmark, the two proposals outperform all the fastest competitors in literature, and therefore represent the new state-of-the-art for connected components labeling on bitonal images.**

***Contribution*—This paper introduces two new Connected Components Labeling algorithms for bitonal images that significantly improve the state-of-the-art, using *Find First Set* instructions.**

***Index Terms*—Connected Components Labeling, Binary Image Processing, Bitonal Images, Algorithms Optimization**

I. Introduction

Connected Components Labeling, or CCL in short, is one of the most important tasks in binary image processing. It consists in assigning a unique label — usually an integer number — to each object, where objects are identified as connected groups of foreground (non-zero) pixels. Several image processing and computer vision applications employ connected components labeling as a pre- or post-processing step [1], [2], [3], [4], [5], [6]. Given the importance of the task, many works have been published in the last two decades that address its runtime optimization, both for sequential [7], [8], [9], [10], [11], [12] and parallel architectures [13], [14], [15], [16], [17], [18].

As regards sequential architectures, the most significant optimizations have proved to be the two-scan approach [19], the use of union-find to resolve label equivalences [20], decision trees [7], [21], the block-based scan mask [22], [23], [24] and state prediction [25], [26].

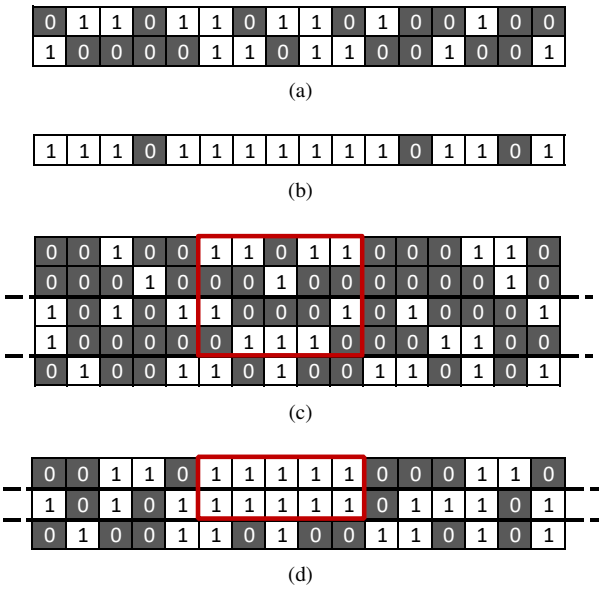


Fig. 1. (a) Example rows to be merged into a single one by means of bitwise *OR*. The image contains four connected components, which corresponds to runs after the merge, whose result is reported in (b). (c) Example image with five rows, that are merged into three rows in (d). Merged runs enclosed in the red rectangle appear to be connected in (d), but original data in (c) show that they are not. Best viewed in color.

The most memory-efficient format for storing binary images uses only 1 bit for each pixel; these are noted as 1bpp images, or *bitonal* images. Systems with limited resources can take the most advantage from this representation; as an example, the United States and many more countries adopted the bitonal format as the legally recognized standard for electronic check clearing. Working with bitonal images has the advantage of considerably reducing the amount of memory accesses, wrt images stored with one byte per pixel. However, the retrieval of single pixel values requires bitwise operations that impact on total execution time.

This paper addresses the problem of connected components labeling on bitonal images. Our work starts from the observation that the combination of the bitonal image format with *Find First Set* instructions allows to efficiently retrieve consecutive blocks of connected pixels, also known as *runs*. *Find First Set* (FFS), also called *Count Leading Zeros*, *Bit Scan Forward* or *Find Leftmost One*, is a bitwise operation that reports the position of the first bit set to 1 in a word, counting from the least significant bit; this is an especially efficient operation because it is implemented in hardware on most recent architectures.

The run-based approach has been applied to CCL before, by He *et al.* [27]. Their proposal, denoted as Run-Based Two-Scan (RBTS), employs the typical two-scan approach, but considers runs as the elementary units of connected components, instead of single pixels or blocks. In this work, RBTS is modified to work with bitonal input, and improved with two more optimizations: FFS instructions, and a more efficient method to store provisional labels. The resulting algorithm is called Bit-Run Two Scan (BRTS).

Then, we noticed that runs computed on the bitwise *OR* between consecutive rows directly correspond to connected components in the original data (Fig. 1). This observation led to the design of another new algorithm, Bit-Merge-Run Scan (BMRS), which finds runs on the bitwise *OR* between pairs of rows, also denoted as *merged runs*. This approach requires a specific method to check connectivity between merged runs, but approximately halves the number of runs to be computed.

The two proposals are evaluated on a public benchmark, and compared to the fastest CCL algorithms available in literature. Experimental results demonstrate that, when fed with bitonal input, BRTS and BMRS outperform all the competitors, establishing the new state-of-the-art for connected components labeling on bitonal images.

The rest of this paper is organized as follows. Section II resumes the most important contributions on connected components labeling; the CCL task is detailed in Section III, alongside a description of the RBTS algorithm; Section IV illustrates the two new proposals, which are evaluated in Section V. Finally, in Section VI conclusions are drawn.

II. Related Work

Originally introduced by Rosenfeld and Pfaltz [19], the labeling of connected components has a very long history, full of different strategies and proposals. Since its first appearance in 1966, many works have shown algorithmic solutions to improve runtime efficiency. Traditionally, the two-scan approach is preferred by the fastest CCL algorithms. In the first scan, each pixel is assigned a specific temporary label using a mask of already visited pixels, and possible equivalences between labels are noted. A representative label is then established for each connected component, and substituted to provisional labels in the second scan.

Various methods have been proposed to address the equivalence of labels, among those the most commonly seen in literature use some variations of union-find. The union-find data structure, first applied to CCL by Dillencourt *et al.* [20], provides two practical procedures for handling equivalence classes: *Find*, which takes the representative label of an equivalence class, and *Union*, which combines two equivalence classes into one.

After the introduction of union-find, Wu *et al.* [7] provided a significant improvement in the form of decision trees, to reduce the average number of load/store operations during the first scan of the input image. The resulting algorithm was called *Scan Array-based Union Find*, or SAUF in short.

Following a totally different approach, He *et al.* [27] published a *run-based* algorithm, which divides connected components in chunks of consecutive foreground pixels (runs); it shares the common two scan structure, but checks connectivity between runs instead of single pixels or blocks. Another run-based strategy is employed by Light Speed Labeling (LSL) [10], which combines it with the Selkow's automaton to reduce the number of labels, and introduces a line-relative labeling to simplify equivalence solving.

In 2010, Grana *et al.* [22] introduced another important step forward, consisting of a 2×2 block-based approach. The problem is modeled as a *command execution metaphor*: the pixel values in the scan mask make up the *rule*, which is linked to a set of equivalent actions in an *OR*-decision table. Given the decision table, the algorithm can easily read all the pixels inside the mask, recognize the rule, and find the action to be performed in the appropriate column. In [21], a dynamic programming approach was proposed to convert *OR* decision tables into optimal binary decision trees, to minimize the average number of pixels to be read when selecting the correct action. The algorithm is called BBDT.

In 2014, He *et al.* [25] proved that the value of already inspected pixels during the horizontal mask shift could be summarized with a finite state machine. In [26], the knowledge of already tested pixels is combined with the optimal decision tree used in [7], resulting in a forest of reduced trees, one for each possible previous pattern, which can be "predicted" in the current mask shift. The algorithm is thus called PRED.

Following another approach, Bolelli *et al.* [28] noticed several equivalent subtrees in the optimal decision tree of [21], and managed to merge them together, obtaining a significant reduction in machine code footprint and increasing instruction cache hit-rate. Because the decision tree was changed into a more generic Directed Rooted Acyclic Graph, the new algorithm is denoted as DRAG. The last improvement of decision tree-based algorithms is represented by Spaghetti [29], where authors managed to combine the block-based mask with state prediction and code compression: the resulting algorithm, known as *Spaghetti Labeling*, was modeled as a Directed Rooted Acyclic Graph with multiple entry points, automatically generated without manual intervention.

III. Preliminaries

In the following, the notation $[\![a, b]\!]$ indicates the set of all *integers* between a and b included:

$$[\![a, b]\!] = [a, b] \cap \mathbb{Z}$$

Be $\mathcal{L} = [\![0, H-1]\!] \times [\![0, W-1]\!]$ a 2D rectangular lattice of H rows and W columns, and $I : \mathcal{L} \to \{0, 1\}$ a binary image. Pixels with value 0 and 1 are respectively said to be *background* ($\mathcal{B}$) and *foreground* ($\mathcal{F}$):

$$\mathcal{F}(I) = \{p \in \mathcal{L} \mid I(p) = 1\}$$

$$\mathcal{B}(I) = \{p \in \mathcal{L} \mid I(p) = 0\}$$

The *8-neighborhood* of a pixel $p = (p_r, p_c)$ is the set

$$\mathcal{N}_8(p) = \{q \in \mathcal{L} \mid max(|p_r - q_r|, |p_c - q_c|) \leq 1\},$$

i.e. the set containing p and the 8 pixels that share a side or a vertex with it, when viewing them as small black or white squares.

Two foreground pixels a and b are said to be *connected* if there is a path of neighboring foreground pixels linking a to b or, more formally:

$$\exists\{p_i \in \mathcal{F}(I) \mid p_1 = a, p_{n+1} = b, p_{i+1} \in \mathcal{N}_8(P_i), i = 1, ..., n\}$$

It is trivial to observe that pixel connectivity is reflexive, symmetric and transitive, and therefore is an equivalence relation, which splits $\mathcal{F}(I)$ in disjoint equivalence classes, also known as *connected components*.

The aim of Connected Components Labeling (CCL) is to define a function $L : \mathcal{L} \to \mathbb{N}_0$, which assigns a unique label to each connected component, reserving label 0 for the whole background.

Most CCL algorithms compute provisional labels for connected components in intermediate steps, before finding the definitive labels which form the final result L. Let $\mathcal{D} : \mathbb{N}_0 \to \mathbb{N}_0$ be the function that associates each provisional label to the corresponding definitive label. Two provisional labels l and m are said to be *equivalent*, $l \equiv m$, if they correspond to the same definitive label at the end of the labeling process, i.e., $l \equiv m \Leftrightarrow \mathcal{D}(l) = \mathcal{D}(m)$. It is easy to observe that this is another equivalence relation, and therefore provisional labels can be split into equivalence classes, each identified by a *representative label* — usually the lowest one in the class. Provisional labels are useful to model the common situation in which two connected regions of the image are initially thought to be separate CCs. When a connection is eventually found, the two provisional labels reveal to be equivalent, and their equivalence classes must be *merged* into one: this operation is known as *equivalence solving*, or *label solving*. Several label solving techniques and data structures exist: the most commonly employed by CCL algorithms are Union-Find (UF), Union-Find with Path Compression (UFPC) [7], Three Table Array (TTA) [8], and interleaved Rem algorithm with SPlicing (RemSP) [30].

Both functions I and L can be represented with multi-dimensional arrays, usually stored in memory in row-major order. The datatype chosen for L is usually the 32-bit integer type, large enough for all the common image sizes. Instead, I could theoretically use just one bit per pixel, even if the most common representation of binary images uses one whole byte per pixel.

A. Run-Based Two-Scan (RBTS)

The first of the two CCL algorithms proposed with this paper, BRTS, is a special optimization of Run-Based Two-Scan (RBTS), the run-based algorithm devised by He *et al.* in [27], which is described in this section.

As the name suggests, the algorithm is built upon the concept of *run*, which is a block of contiguous foreground pixels in a row of $\mathcal{L}$. It is clear that a run is always a subset of a connected component, and that each connected component can be split in a finite set of disjoint runs.

A run starting at pixel $p = (r, c_s)$ and ending with pixel $q = (r, c_e)$ is denoted as $\rho = \mathscr{e}(r, c_s, c_e)$. The *neighborhood of a run* is the union of the neighborhoods of all the pixels in the run:

$$\mathcal{N}_8(\rho) = \bigcup_{p \in \rho} \mathcal{N}_8(p)$$

For $\rho = \mathscr{e}(r, c_s, c_e)$, this corresponds to:

$$\mathcal{N}_8(\rho) = [\![r-1, r+1]\!] \times [\![c_s - 1, c_e + 1]\!] \cap \mathcal{L}$$

Moreover, we define the *upper neighborhood* of a run as the part of the neighborhood in the upper row:

$$\mathcal{N}_8^{\wedge}(\rho) = \{r-1\} \times [\![c_s - 1, c_e + 1]\!] \cap \mathcal{L}$$

The RBTS algorithm performs two scans of the input image I: the first one scans pixels in the raster scan direction, recording data for each run met, assigning provisional labels to runs and recording equivalences; then, the second scan replaces provisional labels with definitive ones.

During the first scan, when a run $\rho = \mathscr{e}(r, c_s, c_e)$ is found, its upper neighborhood is checked for the presence of connected runs, already recorded by the algorithm; $\mathcal{S}(\rho)$ is the set of these runs. Three possibilities can arise for $\mathcal{S}(\rho)$: *(i)* $\mathcal{S}(\rho) = \varnothing$, ρ is assigned a new provisional label; *(ii)* $\mathcal{S}(\rho)$ only contains one run σ, the label of σ is also given to ρ; otherwise, *(iii)* in the case that $\mathcal{S}(\rho)$ contains multiple runs, all of their labels are merged together, and the representative one of the resulting equivalence class is finally assigned to ρ. After the choice of the appropriate label l, the output image L is updated so that $L(p) = l, \forall p \in \rho$.

The second scan replaces the label assigned to each pixel with the representative for the equivalence class, thus completing the labeling process. Optionally, this second scan can be preceded by a *flatten* operation on the label solver, which ensures that definitive labels are consecutive [7].

The first scan requires a method for finding the runs in the upper neighborhood of each new run ρ. In order to accomplish this, run metadata, consisting in start and end coordinates, are recorded in a *run queue*. A run $\sigma = \mathscr{e}(r-1, h, t)$ is connected to $\rho = \mathscr{e}(r, s, e)$ if $h \leq e+1$ and $t \geq s-1$; therefore, when the raster scan reaches run $\rho = \mathscr{e}(r, s, e)$, any run $\sigma = \mathscr{e}(r-1, h, t)$ with $t < s-1$ can no longer be part of the upper neighborhood of any coming run, and its metadata cease to be useful. As a consequence, the run queue can be implemented as a circular buffer of size $(W/2 + 2)$.

Algorithm 1 Runs retrieval algorithm used in BRTS and BMRS, described as a coroutine. Parameters $bits$ and bit_final are pointers to the start and one past the end of the current row, and FFS is the FindFirstSet operation. Word size is assumed to be 64 bits and incrementing a pointer moves it to the next 64 bits.

```
1:  coroutine FINDNEXTRUN(bits, bit_final)
2:      work_bits ← *bits
3:      base ← 0
4:      bitpos ← 0

5:      loop
6:          ▷ Find first non empty word
7:          while FFS(&bitpos, work_bits) = 0 do
8:              bits ← bits + 1
9:              base ← base + 64
10:             if bits ≥ bit_final then        ▷ End of the row
11:                 return (0xFFFF, 0xFFFF)
12:             work_bits ← *bits
13:         start ← base + bitpos
14:         work_bits ← ¬work_bits ∧ (¬0LL ≪ bitpos)

15:         ▷ Find ending position
16:         while FFS(&bitpos, work_bits) = 0 do
17:             bits ← bits + 1
18:             base ← base + 64
19:             if bits = bit_final then
20:                 bitpos ← 0
21:                 work_bits ← ¬0LL
22:                 break
23:             work_bits ← ¬(*bits)
24:         end ← base + bitpos
25:         work_bits ← ¬work_bits ∧ (¬0LL ≪ bitpos)
26:         yield (start, end)
```

IV. METHOD

This paper introduces two new CCL algorithms, specifically designed for dealing with input in bitonal format, i.e., image I is stored with only one bit per pixel, occupying approximately $(H \times W)/8$ bytes. This is the most memory-efficient way to store a binary image, and therefore represents a natural choice of format. The two proposed algorithms are called Bit-Run Two Scan (BRTS) and Bit-Merge-Run Scan (BMRS). In the following, bits in a 64-bit word are supposed to be stored with the leftmost pixel in the least significant bit and the rightmost one in the most significant one. If the image uses an opposite convention, one just needs to exchange Find First Set/Bit Scan Forward instructions with Find Last Set/Bit Scan Reverse ones.

A. Bit-Run Two Scan (BRTS)

This algorithm, as the acronym may suggest, is a special optimization of RBTS, which has been described in the previous section. The basic two-scan structure is inherited, together with the run-based nature and the use of a label solving technique for dealing with equivalence between provisional labels. The main improvement concerns the method to retrieve runs while scanning the input image, that in this case is stored in bitonal format.

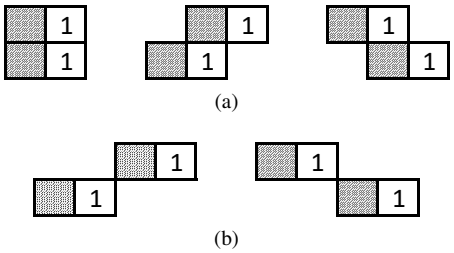


Fig. 2. Examples of connectivity between two foreground pixels belonging to consecutive rows. Foreground pixels in the upper (u) and lower (d) row are denoted by 1, and squares filled with diagonal lines pattern are the added bits in expression $u \vee u \ll 1$ and $d \vee d \ll 1$. The result of $(u \vee u \ll 1) \wedge (d \vee d \ll 1)$ is non-zero for the three cases in (a), which are indeed 8-connected, and is zero for the two non-connected cases in (b).

Start and end position of runs are retrieved by means of *Find First Set (FFS)* instructions. Find First Set is a bitwise operation that, given an unsigned machine word, designates the position of the least significant bit set to one, counting from the least significant bit. Most modern CPU instruction set architectures provide FFS as a hardware operation, making it an efficient way to find start and end positions of runs. The run retrieval procedure is detailed in Algorithm 1.

Another optimization wrt RBTS is that provisional labels are not written in L, until the second scan. Instead, provisional labels are stored in an additional field of run metadata. In this way, each label is written in memory only once per run, instead of once per pixel, during the first scan. The downside of this approach is increased memory occupancy; in fact, run metadata must live until the end of the second scan, and so they cannot be stored in a circular buffer, as in RBTS. Because the worst case is that of alternate foreground and background pixels in each row, the maximum number of runs is $H \times (W/2 + 1)$. Metadata for each run include the the start and end columns, and the provisional label. For most real world applications, coordinates can be stored with 16 bits each and labels require 32 bits, so the total memory requirement for metadata amounts to $H \times (W/2 + 1) \times 8 \approx 4HW$ bytes, the same as the output image L. Total memory allocation for the algorithm is summarized in Table I.

The second scan of BRTS writes definitive labels into L, using the metadata previously saved, and checking the correspondence between provisional and definitive labels with the chosen label solving strategy. Differently from RBTS, this correspondence is only checked once per run, instead of once per pixel; for this reason, this second scan is faster than that of RBTS.

B. Bit-Merge-Run Scan (BMRS)

The second proposal of this paper, Bit-Merge-Run Scan (BMRS), is a modified version of BRTS. In a two-row binary image $I : [\![0, 1]\!] \times [\![0, W - 1]\!] \to \{0, 1\}$, a bitwise *OR* between the upper row and lower row yields connected chunks

Algorithm 2 Junction matrix filling algorithm used by BMRS. Parameters $junc$ and $bits$ are pointers to junction matrix data and image data; h and w are the image dimensions.

procedure FILLJUNCTIONMATRIX($junc, bits, h, w$)
  $h\_merge \leftarrow (h+1)/2$
  $data\_width \leftarrow (w+63)/64$
  for $i \in [\![0, h\_merge-1]\!]$ **do**
    $bits\_u \leftarrow bits + data\_width * (2*i+1)$
    $bits\_d \leftarrow bits + data\_width * (2*i+2)$
    $bits\_dest \leftarrow junc + data\_witdh * i$

    $u\_0 \leftarrow bits\_u[0]$
    $d\_0 \leftarrow bits\_d[0]$
    $bits\_dest[0] \leftarrow (u\_0 \vee (u\_0 \ll 1)) \wedge (d\_0 \vee (d\_0 \ll 1))$
    for $j \in [\![1, data\_width]\!]$ **do**
      $u \leftarrow bits\_u[j]$
      $u\_shl \leftarrow u \ll 1$
      $d \leftarrow bits\_d[j]$
      $d\_shl \leftarrow d \ll 1$
      if $bits\_u[j-1] \wedge (1 \ll 63)$ **then**
        $u\_shl \leftarrow u\_shl \vee 1$
      if $bits\_d[j-1] \wedge (1 \ll 63)$ **then**
        $d\_shl \leftarrow d\_shl \vee 1$
      $bits\_dest[j] \leftarrow (u \vee u\_shl) \wedge (d \vee d\_shl)$

Algorithm 3 Inline function used in BMRS for checking if two merged runs are connected, after having pre-calculated the junction matrix. Parameter $flag\_bits$ points to the junction data between the two lines, s and e mark the starting and ending coordinates of merged runs overlapping.

function ISCONNECTED($flag\_bits, s, e$)
  $s\_base \leftarrow s/64$
  $s\_bits \leftarrow s \bmod 64$
  if $s = e$ **then**
    return $flag\_bits[s\_base] \wedge (1 \ll s\_bits)$
  $e\_base \leftarrow (e+1)/64$
  $e\_bits \leftarrow (e+1) \bmod 64$
  if $s\_base = e\_base$ **then**
    $cutter \leftarrow (\neg 0\text{LL} \ll s\_bits) \oplus (\neg 0\text{LL} \ll e\_bits)$
    return $flag\_bits[s\_base] \wedge cutter$
  for $i \in [\![s\_base+1, e\_base-1]\!]$ **do**
    if $flag\_bits[i]$ **then**
      return true
  $cutter\_s \leftarrow \neg 0\text{LL} \ll s\_bits$
  $cutter\_e \leftarrow \neg(\neg 0\text{LL} \ll e\_bits)$
  if $flag\_bits[s\_base] \wedge cutter\_s$ **then**
    return true
  if $flag\_bits[e\_base] \wedge cutter\_e$ **then**
    return true
  return false

of foreground pixels (runs) that correspond to connected components in I: see the example in Fig. 1a and Fig. 1b.

The whole idea of BMRS is based on this property. A bitwise *OR* is performed for every disjoint pair of consecutive rows, and the result is saved in the *merged input*, which requires approximately $HW/16$ bytes. Then, runs are found on the merged bits; these runs will be referred to as *merged runs*, in opposition to runs computed directly on the raw input.

Unfortunately, checking connectivity between merged runs is not as easy as it is for simple runs. See Fig. 1c and Fig. 1d as an example. The input image has five rows, which result in three merged rows. Merged runs enclosed in the red rectangle of Fig. 1d seem to be connected, but the corresponding connected components are not. Therefore, we need a special method for checking connectivity between merged runs, which must take into account the bits of the original image.

Let $\mathscr{R}(r, c_s, c_e) = [\![r, r+1]\!] \times [\![c_s, c_e]\!]$ be the merged run spanning rows r and $r+1$ and columns from c_s to c_e. Two runs $P = \mathscr{R}(r, s, e)$ and $\Sigma = \mathscr{R}(r-2, h, t)$ must be checked for connectivity if $h \leq e+1$ and $t \geq s-1$. If these conditions are verified, the overlapping segment is delimited between $a = max(s, h) - 1$ and $b = min(e, t) + 1$. Let u and d be respectively the segments of row $r-1$ and r between a and b:

$$u = \{r-1\} \times [\![a, b]\!] \quad (1)$$

$$d = \{r\} \times [\![a, b]\!] \quad (2)$$

As confirmed by the examples in Fig. 2, runs P and Σ are connected iff the following operation gives a non-zero result f, called *junction flag*:

$$f = (u \vee (u \ll 1)) \wedge (d \vee (d \ll 1))$$

Computing junction flags for each pair of runs that may be connected is expensive; therefore, junction flags are pre-calculated for all whole row pairs, and stored in the *junction matrix*, which occupies as much data as the merged input. The algorithm for building the junction matrix is described in Algorithm 2. Then, Algorithm 3 details how junction flags can be used to determine whether two runs are connected; it basically takes the segment of the junction matrix corresponding to their intersection and checks whether it is non-zero.

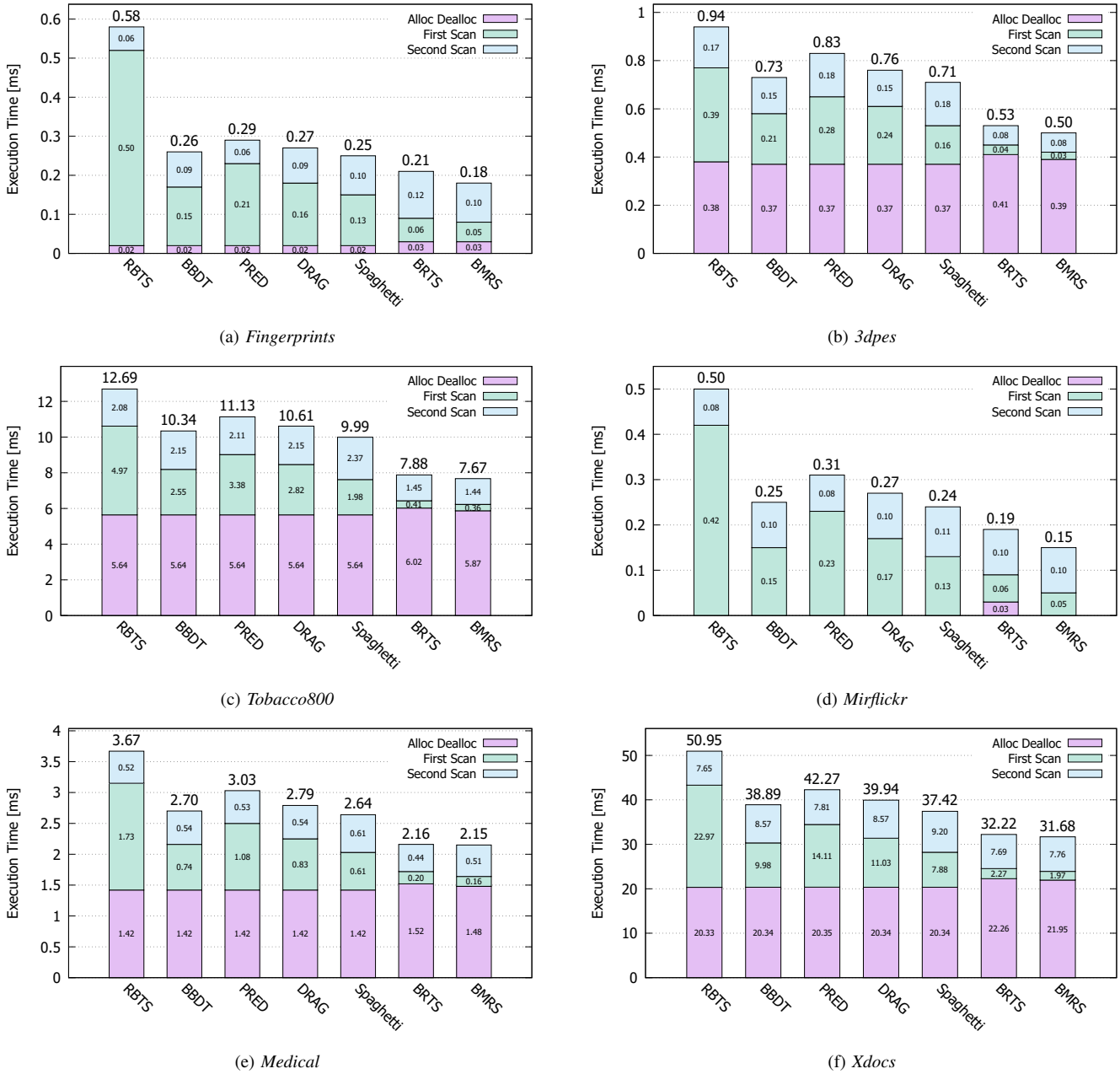
The first scan of BMRS is very similar to that of BRTS, but the input data considered by the raster scan consist of merged rows, instead of the original bitonal image. Connectivity between the current merged run and merged runs in the previous row is checked with the method just described.

The second scan of BMRS is more complex than that of BRTS; in fact, the label of each merged run P must be assigned to all of its foreground pixels, and this requires reading the value of all pixels in the rectangle covered by P, in order to find the foreground.

The memory requirements for both the proposed algorithms are summarized in Table I.

V. EXPERIMENTAL RESULTS

The performance of the proposed algorithms has been evaluated using YACCLAB [28], [31], an open source benchmark-



(a) *Fingerprints*

(b) *3dpes*

(c) *Tobacco800*

(d) *Mirflickr*

(e) *Medical*

(f) *Xdocs*

Fig. 3. Average run-time tests with steps in ms (lower is better) on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. All algorithms employ the UFPC label solver.

ing framework for Connected Components Labeling written in C++. YACCLAB contains a large suite of real world datasets, covering most fields where CCL is usually employed. Experimental results discussed in the following were obtained on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. All the algorithms have been compiled for x64 architecture with optimizations enabled. YACCLAB includes an implementation of all the main CCL algorithms published in recent literature, for comparison with new proposals. BRTS and BMRS have been compared to RBTS, BBDT, PRED, DRAG and Spaghetti, all introduced in Section II. Among those, Spaghetti is the current state-of-the-art. For a fair comparison, each algorithm is provided with input data in its preferred format, i.e., 1-bit-per-pixel for BRTS and BMRS, and 1-byte-per-pixel for all the others. The label solver used by all algorithms is UFPC, which achieves the best performance on average [31].

Fig. 3 reports bar charts of average execution times on real world datasets. Because all the compared algorithms employ a two-scan approach, time measures are divided into memory allocation/deallocation, first scan and second scan, for a more fine-grained comparison. Table II, instead, reports average total times and standard deviations. Some datasets, such as *Tobacco800*, contain images with a significant differ-

TABLE I
MEMORY REQUIREMENT OF BRTS AND BMRS, COMPARED TO THAT OF SPAGHETTI. VALUES ARE EXPRESSED IN BYTES PER PIXEL, SO THE TOTAL AMOUNT CAN BE OBTAINED MULTIPLYING THE VALUES BY $W \times H$.

	Spaghetti	BRTS	BMRS
Output Image	4	4	4
Label Solver	1	1	1
Run Metadata	-	4	2
Merged Input	-	-	1/16
Junction Matrix	-	-	1/16
Total	5	9	7+1/8

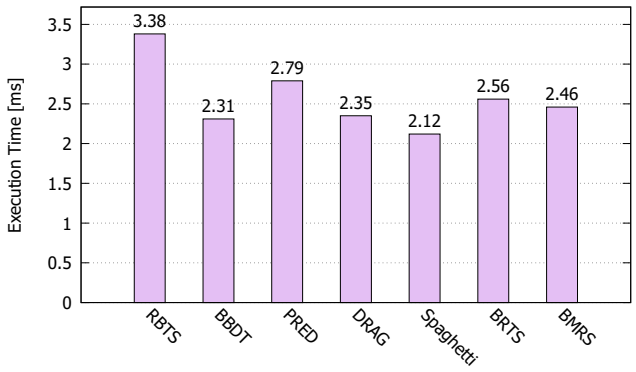


Fig. 4. Average run-time tests on the *Medical* dataset. The input image is 1-byte-per-pixel for all the algorithms: times of BRTS and BMRS include the conversion to 1-bit-per-pixel format. Measured on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. All algorithms employ the UFPC label solver.

ence in resolution, causing high standard deviation in CCL performance. In all datasets, RBTS is by far the slowest algorithm; BBDT, PRED, DRAG and Spaghetti have similar performance, with Spaghetti being always the best of the four; BRTS and BMRS outperform all the competitors, and BMRS is always the fastest. In particular, the speedup of BRTS and BMRS wrt Spaghetti ranges from 1.16 to 1.34 and from 1.18 to 1.60 respectively. The overall improvement is greater on datasets of small images (Fingerprints and Mirflickr), where alloc/dealloc has the lowest impact on total execution time. Comparing the original RBTS algorithm to BRTS, it is clear that the first scan is the most optimized step, with an average speedup of 9.3. This huge performance improvement is due to the introduction of the FFS operation, which is available as a hardware instruction on 64-bit words on the selected test system, as long as in most modern CPU ISAs. The same optimization also make BRTS faster than state-of-the-art algorithms, which do not employ specific hardware operations or compiler intrinsics. The other proposal, BMRS, furtherly optimizes the first scan, approximately halving the amount of runs to be scanned and processed. Moreover, in spite of using two more data structures, the total memory needed by BMRS is actually less than that required by BRTS, allowing a little alloc/dealloc time saving.

Both the two proposals require more data structures than state-of-the-art algorithms: the total amount of allocated memory is, for BRTS, almost twice that of Spaghetti, as can be seen in Table I. However, the extra memory needed is only equal to that of the output, and therefore does not represent an issue on most systems. Moreover, as demonstrated by the experiments, the impact of these additional allocations on execution time is minimal.

Another possible downside of the new proposals is their preferred bitonal input format. This can raise issues when working with libraries which do not natively support it, such as OpenCV as of today (release 4.5.2). If the input provided to BRTS or BMRS is not bitonal, both algorithms must begin with a conversion. Fig. 4 depicts average execution times on the Medical dataset in this case: the input is 1-byte-per-pixel for all the algorithms. As can be observed, BRTS and BMRS perform worse than current state-of-the-art. However, the bitonal format is the most memory-efficient representation of binary images, and as such there are also some computer vision libraries that natively support it. An example is Leptonica, an open-source image processing library employed in several projects (e.g. tesseract OCR by Google). When the input is in bitonal format, BRTS and especially BMRS are the CCL algorithms of choice.

VI. CONCLUSION

Two new run-based CCL algorithms have been presented, which employ Find First Set instructions to efficiently retrieve runs from a bitonal input. Experiments conducted over a collection of real world datasets demonstrate that both proposals outperform the fastest CCL algorithms in literature, thus representing the new state-of-the-art for connecting components labeling on bitonal images. The source code is available in [32].

REFERENCES

[1] F. Uslu and A. A. Bharath, "A recursive Bayesian approach to describe retinal vasculature geometry," *Pattern Recognition*, vol. 87, pp. 157–169, 2019.

[2] I. H. Laradji, N. Rostamzadeh, P. O. Pinheiro, D. Vazquez, and M. Schmidt, "Where are the Blobs: Counting by Localization with Point Supervision," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 547–562.

[3] F. Pollastri, F. Bolelli, R. Paredes, and C. Grana, "Augmenting Data with GANs to Segment Melanoma Skin Lesions," *Multimedia Tools and Applications*, vol. 79, no. 21-22, pp. 15 575–15 592, May 2019.

[4] H. V. Pham, B. Bhaduri, K. Tangella, C. Best-Popescu, and G. Popescu, "Real time blood testing using quantitative phase imaging," *PloS one*, vol. 8, no. 2, p. e55676, 2013.

[5] L. Canalini, F. Pollastri, F. Bolelli, M. Cancilla, S. Allegretti, and C. Grana, "Skin Lesion Segmentation Ensemble with Diverse Training Strategies," in *International Conference on Computer Analysis of Images and Patterns*. Springer, 2019, pp. 89–101.

[6] F. Bolelli, G. Borghi, and C. Grana, "XDOCS: An Application to Index Historical Documents," in *Italian Research Conference on Digital Libraries (IRCDL)*. Springer, 2018, pp. 151–162.

[7] K. Wu, E. Otoo, and K. Suzuki, "Two Strategies to Speed up Connected Component Labeling Algorithms," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-59102, 2005.

[8] L. He, Y. Chao, and K. Suzuki, "A Linear-Time Two-Scan Labeling Algorithm," in *International Conference on Image Processing*, vol. 5, 2007, pp. 241–244.

TABLE II

AVERAGE RUN-TIME TESTS IN MS, ± STANDARD DEVIATION. RESULTS HAVE BEEN OBTAINED ON AN INTEL(R) CORE(TM) I7-4790 CPU @ 3.60GHZ WITH WINDOWS 10.0.17134 (64 BIT) OS AND MSVC 19.15.26730 COMPILER. ALL ALGORITHMS EMPLOY THE UFPC LABEL SOLVER. NOVEL PROPOSALS ARE MARKED WITH A STAR.

	RBTS	BBDT	PRED	DRAG	Spaghetti	BRTS*	BMRS*
Fingerprints	0.58 ± 0.20	0.26 ± 0.13	0.29 ± 0.14	0.27 ± 0.13	0.25 ± 0.12	0.21 ± 0.11	$\mathbf{0.18} \pm 0.09$
3dpes	0.94 ± 0.17	0.73 ± 0.05	0.83 ± 0.06	0.76 ± 0.06	0.71 ± 0.05	0.53 ± 0.08	$\mathbf{0.50} \pm 0.06$
Tobacco800	12.69 ± 7.78	10.34 ± 6.42	11.13 ± 6.90	10.61 ± 6.60	9.99 ± 6.20	7.88 ± 4.91	$\mathbf{7.67} \pm 4.87$
Mirflickr	0.50 ± 0.26	0.25 ± 0.80	0.31 ± 0.10	0.27 ± 0.08	0.24 ± 0.08	0.19 ± 0.12	$\mathbf{0.15} \pm 0.06$
Medical	3.67 ± 1.54	2.70 ± 1.14	3.03 ± 1.27	2.79 ± 1.18	2.64 ± 1.12	2.16 ± 0.96	$\mathbf{2.15} \pm 0.95$
Xdocs	50.95 ± 5.39	38.89 ± 4.26	42.27 ± 4.66	39.94 ± 4.37	37.42 ± 4.11	32.22 ± 3.46	$\mathbf{31.68} \pm 3.70$

[9] F. Bolelli, L. Baraldi, M. Cancilla, and C. Grana, "Connected Components Labeling on DRAGs," in *International Conference on Pattern Recognition (ICPR)*. IEEE, 2018, pp. 121–126.

[10] L. Lacassagne and B. Zavidovique, "Light speed labeling: efficient connected component labeling on risc architectures," *Journal of Real-Time Image Processing*, vol. 6, no. 2, pp. 117–135, 2011.

[11] F. Bolelli, S. Allegretti, and C. Grana, "One DAG to Rule Them All," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–12, 2021.

[12] L. He, X. Ren, Q. Gao, X. Zhao, B. Yao, and Y. Chao", "The connected-component labeling problem: A review of state-of-the-art algorithms," *Pattern Recognition*, vol. 70, pp. 25–43, 2017.

[13] S. Zavalishin, I. Safonov, Y. Bekhtin, and I. Kurilin, "Block Equivalence Algorithm for Labeling 2D and 3D Images on GPU," *Electronic Imaging*, vol. 2016, no. 2, pp. 1–7, 2016.

[14] S. Allegretti, F. Bolelli, M. Cancilla, F. Pollastri, L. Canalini, and C. Grana, "How does Connected Components Labeling with Decision Trees perform on GPUs?" in *Computer Analysis of Images and Patterns*. Springer, September 2019, pp. 39–51.

[15] D. P. Playne and K. Hawick, "A new algorithm for parallel connected-component labelling on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1217–1230, June 2018.

[16] S. Allegretti, F. Bolelli, M. Cancilla, and C. Grana, "Optimizing GPU-Based Connected Components Labeling Algorithms," in *Third IEEE International Conference on Image Processing, Applications and Systems (IPAS)*, 2018.

[17] ——, "A Block-Based Union-Find Algorithm to Label Connected Components on GPUs," in *Image Analysis and Processing - ICIAP 2019*, 2019, pp. 271–281.

[18] S. Allegretti, F. Bolelli, and C. Grana, "Optimized Block-Based Algorithms to Label Connected Components on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, pp. 423–438, August 2019.

[19] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital picture processing," *Journal of the ACM*, vol. 13, no. 4, pp. 471–494, Oct. 1966.

[20] M. B. Dillencourt, H. Samet, and M. Tamminen, "A General Approach to Connected-Component Labeling for Arbitrary Image Representations," *Journal of the ACM*, vol. 39, no. 2, pp. 253–280, 1992.

[21] C. Grana, M. Montangero, and D. Borghesani, "Optimal decision trees for local image processing algorithms," *Pattern Recognition Letters*, vol. 33, no. 16, pp. 2302–2310, 2012.

[22] C. Grana, D. Borghesani, and R. Cucchiara, "Optimized Block-based Connected Components Labeling with Decision Trees," *IEEE Transactions on Image Processing"*, vol. 19, no. 6, pp. 1596–1609, 2010.

[23] W.-Y. Chang and C.-C. Chiu, "An efficient scan algorithm for block-based connected component labeling," in *22nd Mediterranean Conference of Control and Automation (MED)*. IEEE, 2014, pp. 1008–1013.

[24] W.-Y. Chang, C.-C. Chiu, and J.-H. Yang, "Block-based connected-component labeling algorithm using binary decision trees," *Sensors*, vol. 15, no. 9, pp. 23 763–23 787, 2015.

[25] L. He, X. Zhao, Y. Chao, and K. Suzuki, "Configuration-Transition-Based Connected-Component Labeling," *IEEE Transactions on Image Processing"*, vol. 23, no. 2, pp. 943–951, 2014.

[26] C. Grana, L. Baraldi, and F. Bolelli, "Optimized Connected Components Labeling with Pixel Prediction," in *Advanced Concepts for Intelligent Vision Systems (ACIVS)*. Springer, 2016, pp. 431–440.

[27] L. He, Y. Chao, and K. Suzuki, "A Run-Based Two-Scan Labeling Algorithm," *IEEE Transactions on Image Processing*, vol. 17, no. 5, pp. 749–756, 2008.

[28] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, "Towards reliable experiments on the performance of Connected Components Labeling algorithms," *Journal of Real-Time Image Processing*, vol. 17, no. 2, pp. 229–244, February 2018.

[29] F. Bolelli, S. Allegretti, L. Baraldi, and C. Grana, "Spaghetti Labeling: Directed Acyclic Graphs for Block-Based Connected Components Labeling," *IEEE Transactions on Image Processing*, pp. 1999–2012, October 2019.

[30] E. W. Dijkstra, *A discipline of programming*. Prentice-Hall Englewood Cliffs, N.J, 1976.

[31] C. Grana, F. Bolelli, L. Baraldi, and R. Vezzani, "YACCLAB - Yet Another Connected Components Labeling Benchmark," in *2016 23rd International Conference on Pattern Recognition (ICPR)*. Springer, December 2016, pp. 3109–3114.

[32] The YACCLAB Benchmark. Accessed on 2021-06-23. [Online]. Available: https://github.com/prittt/YACCLAB