Journal of RealTime Image Processing manuscript No.
(will be inserted by the editor)

Lionel Lacassagne · Bertrand Zavidovique

Light Speed Labeling

Efficient Connected Component Labeling on RISC Architectures

Received: date / Revised: date

Abstract This article introduces two fast algorithms for Connected Component Labeling of binary images, a peculiar case of coloring. The first one, *Selkow*$_{DT}$ is pixel-based and a Selkow's algorithm combined with the Decision Tree optimization technique. The second one called *Light Speed Labeling* is segment-based *line-relative* labeling and was especially thought for commodity RISC architectures. An extensive benchmark on both structured and unstructured images substantiates that these two algorithms, the way they were designed, run faster than Wu's algorithm claimed to be the world fastest in 2007. Also they both show greater data independency hence runtime predictability.

Keywords

Connected Component Labeling, run length labeling, line relative labeling, Algorithm Architecture Adequation, Rosenfeld, Selkow, Real-Time implementation, transitive closure computation.

Introduction

Binary Connected Component Labeling (CCL) algorithms are widely used in the Image Processing field (Fig. 1). They belong to a wider class of problems in the Graph Theory area and deal with graph coloring and transitive closure computation. CCL algorithms play a central part in machine vision, because they often constitute a mandatory step between low-level image processing (filtering) and high-level image processing (recognition, decision). As such, CCL algorithms have a lot of applications and derivate algorithms like convex hull computation, hysteresis filtering or geodesic reconstruction.

In its most common version, CCL is completed by two coupled finite state automaton running at the same location p respectively on the initial image (data) and the result image (labels). Both automatons transform a common set of neighbor pixels, the predecessors along the image scan, into the label of p depending on their value as a data and their attributed label. Due to its limited horizon, such an automaton artificially generates multiple labels for a given region – e.g. in cases of a concavity (Fig. 3) – to be noticed at the bottom of the concavity and resolved at the end of the image scan. CCL can address pixel sets to 1 (objects or regions out of convention) or, concurrently, both pixel sets to 1 (objects) and zero (background out of convention).

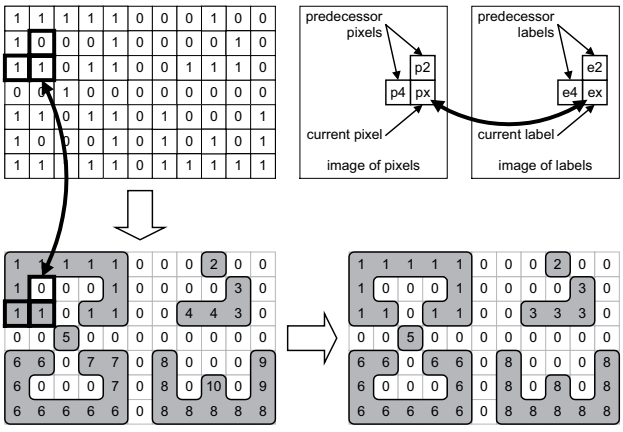

Fig. 1 Example of 4-connected binary components labeling

Due to their importance in vision, a lot of CCL algorithms have been developed in the past. Some typical ones are described in the following sections. Designing a new algorithm is then a challenging task both considering the overwhelming literature and from the very performance of best existing algorithms. It is comparable to developping a

Lionel Lacassagne
Institut d'Electronique Fondamentale (IEF/AXIS)
Université Paris Sud
E-mail: lionel.lacassagne@u-psud.fr

Bertrand Zavidovique
Institut d'Electronique Fondamentale (IEF/AXIS)
Université Paris Sud
E-mail: bertrand.zavidovique@u-psud.fr

new version of matrix multiplication. Goals could be a faster algorithm on some given class of computer architecture or minimizing the amount of memory used, it could be as well to minimize the number of over-created labels or to show the smallest theoretical complexity (not quite the same as to be the fastest one). Yet another issue is to be most predictable.

Now, from the current state of the computing technology, reaching decent performances in actuality requires for CCL algorithms to take into account two specificities/capacities of RISC architectures: the processor pipeline and its cache memories. That amounts to minimize conditional statements (like tests and comparisons) to reduce the number of pipeline stalls and limit random sparse (typically vertical) memory accesses, to lower cache misses.

The first section describes Historical algorithms written by the pioneers, and some Modern algorithms that try to optimize the previous ones. For each of them, we focus on its hardware advantages and drawbacks. This section also describes a forgotten algorithm: the Selkow's algorithm that is used to replace the commonly used Union-Find structure and algorithm in managing equivalence creation. A fair part of these tentative improvements can be framed into an eventual pixel-based CCL algorithm that is an *hybrid* of architectural optimizations presented in the section.

Then we introduce our new algorithm called *Light Speed Labeling* (LSL) specifically designed in view of RISC architectures. This algorithm uses a segment approach combined with the Selkow's algorithm, to minimize the number of created labels. Its major improvement is the introduction of a new *line-relative* labeling to simplify equivalence building between segments.

Finally, some *extensive* benchmarks are run to compare and evaluate a dozen of CCL algorithms. At first we priviledge the sole execution point of view. There, we put a fair stress on the statistical standard deviation of the algorithm execution-time when processing random and quite unstructured images.

But an other important point to consider when designing a CCL algorithm is its goal. As it is an intermediate level algorithm, it processes the output data coming from low level algorithms (filtering, binary segmentation, ...) and provides abstract input data to other intermediate or high level (decision) algorithms. Usually, such abstract data also called *features* are the boundary of bounding rectangle (for target tracking) and the first order statistical moments (surface, centroid, orientation, ...). So if a standalone CCL algorithm can be considered at first step, the couple "CCL + feature computation" is the procedure to be actually evaluated at end. Whence benchmarking on real images – OCR, cadastre and less regular natural images stored in Sidba, Waterloo or Brodatz databases – and rather stressing the distance between actual and optimal execution.

1 Classical labeling algorithms

We present here two sets of algorithms: the historical ones that were designed by pioneers in the field and the modern ones that aim at optimizing the first set.

1.1 Definitions

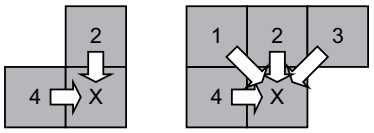

Fig. 2 4 and 8-connected component labeling

Let us define some notations (Fig. 2):

- p_x, e_x, the current pixel and its label
- p_1, p_2, p_3, p_4, the neighbor pixels
- e_1, e_2, e_3, e_4, the associated labels
- ne, the number of labels
- T, the equivalence table
- a, the ancestor (the primal equivalence label) of e: $a = T[e]$
- na, the number of labels (ancestors) after packing T
- *run*: a set of pixels, on a line, with same label.

1.2 Historical algorithms

Algorithm 1: Find algorithm

Input: e a label, T an equivalence table
Result: a, the ancestor of e

```
1 a ← e
2 while T[a] ≠ a do
3     a ← T[a]
4 return a
```

Algorithm 2: Union algorithm

Input: e_1, e_2 two labels, T an equivalence table
Result: a, the least common ancestor of the e's

```
1 a1 ← find(e1, T)
2 a2 ← find(e2, T)
3 if a1 < a2 then
4     a ← a1, T[a2] ← a
5 else
6     a ← a2, T[a1] ← a
7 return a
```

Algorithm 3: positive min value min^+

```
Input: e1, e2, e3, e4 four labels with at least one non-zero label
Result: m, the smallest non zero value
m ← +∞
foreach e_k ∈ {e1, e2, e3, e4} do
    if (e_k ≠ 0 and e_k < m) then m ← e_k
return m
```

1.2.1 Rosenfeld

Algorithm 4: First scan of Rosenfeld's algorithm

```
Input: e1, e2, e3, e4, four labels
foreach pixel p_x do
    if p_x ≠ 0 then
        if (e1 = e2 = e3 = e4 = 0) then
            ne ← ne + 1
            e_x ← ne
        else
            foreach e_k ∈ {e1, e2, e3, e4} do
                a_k ← Find(e_k, T)
            e_x ← min+(a1, a2, a3, a4)
            foreach e_k ∈ {e1, e2, e3, e4} do
                if (a_k ≠ 0 and a_k ≠ e_x) then
                    Union(e_x, a_k, T)
    else
        e_x ← 0
```

The most popular algorithm is Rosenfeld's one (22). He had introduced an algorithm (Algo. 4) which only requires two scans of the image. The first step creates a new label for each newly encountered region (that is a set of connected pixels). The bright idea was to store equivalences between labels into a table T, then to update equivalences before to solve them (transitive closure). First implementations were based on an *adjacency* (aka *incidence*) matrix and the Floyd-Warshall algorithm was used for transitive closure. Their major drawback was the huge amount of memory required to hold the adjacency matrix, its size being the square of the number of labels. The next implementations were based on the Union-Find Algorithms (Algo. 1 & 2) designed by Tarjan. See (8) for details, optimizations and references. Same as transitive closure, Union-Find stems from the general Graph Theory and is not dedicated to image processing. It is an algorithm and a data structure for disjoint sets that allows to compute the transitive closure with efficient complexity. In simplifying the transitive closure (Algo. 5), the positive minimum value (Algo. 3) is propagated during the equivalence building. The equivalence table T is then solved (transitive closure) The algorithm 5 does not produce contiguous numbers as labels, but it can be modified to pack labels on the fly (Algo. 6). Then the image of labels is updated (Algo. 7).

Algorithm 5: equivalences resolution

```
for e ∈ [1 : ne] do
    T[e] ← T[T[e]]
```

Algorithm 6: equivalence resolution & pack

```
for e ∈ [1 : ne] do
    if T[e] ≠ e then
        T[e] ← T[T[e]]
    else
        na ← na + 1
        T[e] ← na
```

Algorithm 7: Second scan of Rosenfeld's algorithm

```
Input: E, image of labels
foreach label e_x ∈ E do
    e_x ← T[e_x]
```

1.2.2 Haralick

The next algorithm is from Haralick (10) and is the classical iterative or multi-pass algorithm. This "step back" was driven by an architecture constraint: the memory required to hold the equivalence table was exceeding computer capacities at the time (Haralick was experimenting on a VAX-11). Accesses to the table were generating as many accesses to the mass storage, making the two-pass algorithm slower than the multi-pass one. Here is a perfect example where the theorical complexity was *diverging* from the computing time.

1.2.3 temporary-label problem: Lumia & Ronse answers

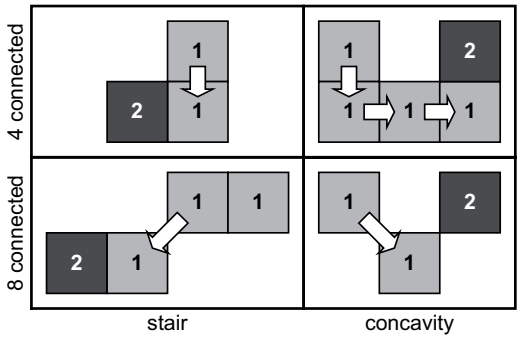

Fig. 3 4 and 8-connected basic patterns: stair and concavity

Only two basic patterns trigger label creation, whatever the connexity. The first one is the *concavity*. From the neighborhood that founds CCL, it is obvious that the label creation can not be avoided. The second one is the *stair*. It is responsible for the burst of unnecessary created labels in the Rosenfeld's algorithm. In figure 3, arrows indicate the label prop-

agation from label to label in light gray while the creation of a new label is figured with dark gray. The two following algorithms provide solutions to avoid these creations.

The Lumia's algorithm (16) is a worthy solution to limit the amount of created labels. For every line, a small equivalence table monitors the label creation and stores equivalences detected along this line. At the end of line, the equivalences are solved for the small table and propagated to the large equivalence table. Such a strategy saves labels, since a new label is not created when a pixel is detected without connexion in its neighborhood (according to Fig. 3). The creation is postponed to an hypothetic adjacency that could happen (or not) between the following pixel of the line and the previous line.

An alternative solution to the large amount of labels is proposed by Ronse (21). It consists in considering segments instead of pixels. Segment labeling makes the number of created labels optimal i.e. equal to the number of ancestors plus the number of concavities:

$$ne = na + nc \tag{1}$$

Algorithm 8: segment algorithm: *implicit* version

```
1  foreach segment s_k^i ∈ S^i, the set of segments on line i do
2      find S^{i-1}(s_k^i) the subset of adjacent segments of S^{i-1} on
       line i − 1;
3      foreach segment s_l^{i-1} ∈ S^{i-1}(s_k^i) of label e_l do
4          apply Union-Find algorithm to find ε the min of all e_l
5          set the label of s_k^i equal to ε
```

Remark: Note that the type of an algorithmic expression (Algo. 8) maybe misleading. It should be understood that the elegant contraction of the lines 3 to 5 covers actually a sequence of conditional statements (Algo. 9) that make all the complexity of the problem, almost the key-point of the present paper and our motivation to design a novel algorithm (LSL cf. section 2). In the present paper, for sake of technical comparison, we introduce a distinction between the algorithm and its actual implementation, possibly with optimizations, so called *procedure*.

1.3 Modern algorithms

1.3.1 Modern pixel-based algorithms

All modern algorithms, except those of section 1.4, derive from one of the historical ones. They try improvement by replacing some components by a more efficient one. For instance the Haralick's algorithm can produce *more efficient* ones in two ways: smaller theoretical complexity or shorter execution time. The latter is definitely favored through the choice Decision Tree vs. Path Compression.

Algorithm 9: segment algorithm: *explicit* version

```
1  foreach segment s_k^i ∈ S^i, the set of segments on line i do
2      find S^{i-1}(s_k^i) the subset of adjacent segments of S^{i-1} on
       line i − 1;
3      if card(S^{i-1}(s_k^i)) = 0 then
4          ne ← ne + 1
5          e_x ← ne
6      else
7          ε ← label of first segment of S^{i-1}(s_k^i)
8          remove first segment from S^{i-1}(s_k^i)
9          foreach segment s_l^{i-1} ∈ S^{i-1}(s_k^i) of label e_l do
10             Apply Union-Find to e_l and ε
11         e_x ← ε
```

Path Compression (PC) as defined in (8) is a second step added to the Union-Find algorithm to perform a transitive closure in climbing up to a common ancestor.

It is probably one of the best examples to illustrate that relying on theoretical complexity is hazardous. Indeed, it is proven (8) that implementing PC with Union-Find makes complexity to grow as slow as the reverse Ackermann function. The latter property is definitively important for graph merging in the general case, but not for equivalence management in CCL according to benchmarks and observations (section 4.3).

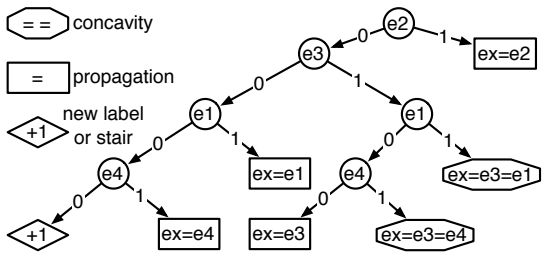

Fig. 4 8-connected Decision Tree: pixel topology that corresponds respectively to a stair, a propagation, a concavity is indicated by a diamond, a rectangle, an octagon

Decision Tree (DT) is a dedicated optimization to CCL. DT uses the local topology at the current pixel (Fig. 4) to reduce the number of pixels to read for finding out the current label. It has a deep impact on the speed of CCL as it tackles one of the processor-architecture problem: pipeline stalls due to conditional instructions. In the classical 8-connected Rosenfeld's algorithm (Algo. 4), there are four tests to compute the minimal positive value of four labels (Algo. 3) and four other tests whether to call the Union algorithm. Note that tests are actually mandatory to avoid merging a label with a 0-value *background* label. With DT (Fig. 4), the average number of tests drops to only 2.25 (1 + 1/2 + 1/4 + 1/4 + 1/8 + 1/8). A single equal in a leaf means a call to the Find algorithm while a double equal means an equivalence building with a call to the Union algorithm. The leaf *new* means

the creation of a new label. The DT has been used by Wu (28) in a 8-connected version, by Lamathy and Demigny in a 4-connected version implemented on a FPGA (15) and by Stepahno and Bulgarelli targetting a General Purpose Processor (26). Depending on the architecture, DT provides a speedup from $\times 1.15$ to $\times 1.50$.

Algorithm 10: Montanvert's version of the Rosenfeld's algorithm (Algo. 4)

```
   Require: T[e] = e,   ne = 0
 1 foreach pixel p do
 2     if p_x ≠ 0 then
 3         if (e_1 = e_2 = e_3 = e_4 = 0) then
 4             new label
 5             ne ← ne + 1
 6             e_x ← ne
 7         else
 8             foreach e_k ∈ {e_1, e_2, e_3, e_4} do
 9                 a_k ← T[e_k]
10             e_x ← min^+(a_1, a_2, a_3, a_4)
11             update a_k:
12             foreach a_k ∈ {a_1, a_2, a_3, a_4}, with a_k ≠ e_x do
13                 while T[a_k] ≠ e_x do
14                     m ← T[a_k]
15                     T[a_k] ← e_x
16                     a_k ← m
17     else
18         e_x ← 0
```

The optimization by Montanvert (7) was to fuse the Find and the Union steps together. Yet, this algorithm implements Path Compression (Algo. 10, lines 12-14).

The optimization by Suzuki is also interesting (27) as it was taylored for an efficient hardware (FPGA or ASIC) implementation. It is an hybrid multi-pass algorithm with forward and backward scans. The authors show that only four passes are required over the tested image bases. A more extensive bibliography can be found in (11) and (28).

1.3.2 Modern segment-based algorithms

After Ronse, many routines to speed up segment-based CCL algorithms were tried too. Recent algorithms focus on algorithm/architecture adequacy.

Yang (29) relies on the *Line Scan Clustering* technique. His algorithm remains slow because of the amount of information associated to each segment, but its major drawback is its hyper sensitivity to data: the execution time can be multiplied by a factor as large as $\times 4$.

He (11) bets on linked lists of segments implemented as a 1-dimentional array. This smart implementation boosts the execution time (by reducing the time spent into equivalence building) and makes the procedure as fast as the *world fastest* algorithm by Wu (28).

Most of the concerned algorithms rely on *Run Length Coding* (RLC). For each segment, a 4- or 8-connected segment is searched in the previous line. This algorithm behaves like a fusion sort. Managing equivalences can be done in the Rosenfeld or Selkow style. Very short segments represent the worst case for this RLC algorithm. The equivalences are processed from left to right, and segment labeling is completed in parallel. So the algorithm is simpler.

1.3.3 Selkow's algorithm: the forgotten algorithm

The Selkow's algorithm (23), quite forgotten, comes from Graph Theory as well to replace the Union-Find algorithm. To our knowledge it never spread over the Image Processing Community except in France where it was used systematically since the early eighties at the ETCA – a laboratory part of the french DARPA – where this algorithm belongs to the Culture (1) (30).

It secures a double access to the equivalence table T (Algo. 11). In most images this is enough to accessing the root label of the graph, that is here the ancestor of the equivalence class. In such a case (98.14 % Fig. 16), no more loop is required (Algo. 1, line 2) so there is no more *pipeline stall* unlike for the Union-Find algorithm.

Algorithm 11: Selkow algorithm

```
   Input: e_1, e_2, e_3, e_4, four labels
 1 foreach pixel p_x do
 2     if p_x ≠ 0 then
 3         if (e_1 = e_2 = e_3 = e_4 = 0) then
 4             ne ← ne + 1
 5             e_x ← ne
 6         else
 7             foreach e_k ∈ {e_1, e_2, e_3, e_4} do
 8                 a_k ← T[T[e_k]]
 9             e_x ← min^+(a_1, a_2, a_3, a_4)
10             update a_k:
11             foreach a_k ∈ {a_1, a_2, a_3, a_4} do
12                 if (a_k ≠ 0 and a_k ≠ e_x) then T[a_k] ← e_x
13     else
14         e_x ← 0
```

Unfortunately, in terms of graph breaking, the Selkow-based algorithm is not a panacea. It is easy to build counter-examples of bristling-enough concavities such that the equivalence will be lost or not (Fig. 5). However the histogram (Fig. 16) of equivalence breaks vs. pixel density shows that the double access Selkow's algorithm overcomes 98 % of the cases of our most stressing data, the *percolation* benchmark. Conversely, in a given application where the structure of concavities is predictable – e.g. target tracking – the access number of Selkow can be adapted. Such hardware inspired a consideration is similar to Suzuki's (27).

As for Path Compression, its execution time relates directly to the depth of the equivalence tree. Results in table 4

indicate that whatever the type of data, it appears a significant correlation between the number of labels ($\simeq$ number of ancestors, concavities, stairs – see caption of Fig. 4) and the latter depth.

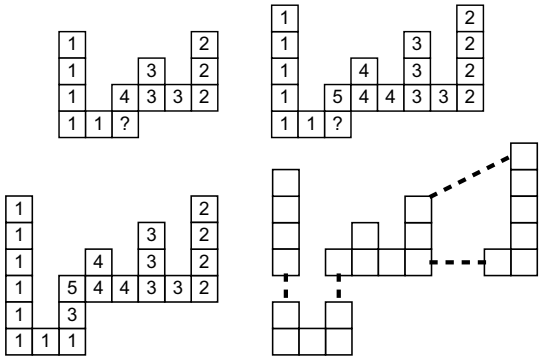

Fig. 5 Line 1: smallest known patterns requiring double access and triple access for pixel-based Selkow's algorithm (4-connected component labeling). Line 2: left: counter-example (although similar this pattern does not cause any graph break) right: generalization pattern where the depth of the first concavity and the number of columns are linked

1.4 Aesthetic but a priori inefficient algorithms

There are at least two species of aesthetic algorithms. The first is based on stack manipulation and the second on the Freeman's code.

1.4.1 Mathematical Morphology

Mathematical morphology usually asks for stacks in the implementation of morphological operators, including CCL. The algorithm here looks like the *painter* algorithm where calls to a recursive function are replaced by a user-defined stack that holds coordinates of every pixel pushed into the stack (25).

1.4.2 Contour Tracking

Contour Tracking (6) may be the most aesthetic optimization as it is a one-pass algorithm where features like the binding box can be computed without labeling any pixel strictly inside the area. Contour Tracking is enough to compute binding boxes or geometrical moments. It is like getting the result of a computation before to finish reading the input. Such a kind of contour algorithm serves in 3D medical imaging too (18).

But, in isolation, it is likely not to run fast as it does not account for specificities of current architectures. Because of the memory hierarchy inside commodity used RISC processors, an algorithm performing vertical or quasi-random accesses instead of horizontal and systematic accesses will hamper the memory caches. The problem is the same as the one already mentioned for Path Compression: PC can turn out to be slower than the Selkow's algorithm that always performs with two memory accesses only.

1.4.3 Cycle equivalence management

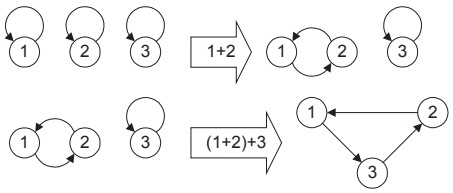

Fig. 6 cycle equivalence management

We tried (13) an alternative strategy to the Union-Find algorithm and data structure that is implementing the equivalence relation as a directed cyclic graph (Fig. 6). But merging graphs together is unfortunately not so straight forward: linking *predecessor's* and *successor's* labels of both graphs is not enough when the two labels belong to the same graph. In some cases, such a link splits the graph into two subgraphs. One has to check if labels are already equivalent (that happens when there are holes in the components) by scanning the whole graph. Both theoretical complexity and execution time grow with the size of the equivalence classes.

1.5 Conclusion

On the one hand, there is the Union-Find with Path Compression algorithm whose complexity grows as the inverse of the Ackermann function and on the other hand, there is the Selkow's algorithm with constant complexity of 2.

The Selkow's algorithm behavior is very close to the Rosenfeld's one. When optimized with DT, we will show that Selkow+DT always run as fast as Rosenfeld+DT+PC Wu's algorithm. But on a commodity single RISC processor, Selkow's algorithm always runs faster (whatever the nature of considered images). Moreover, it is *less sensitive* to data: the execution time standard deviation is always smaller, indicating a more runtime predictable algorithm. Such a property is a key feature for algorithms used in embedded systems and, beyond, on parallel machines (load balancing). Let us call Selkow$_{DT}$ the Selkow's algorithm optimization with DT added that we advocate for, in the pixel-based category.

Concerning segment-based algorithms, the label is attributed to a segment after the equivalence process, not during it. For that reason, the problem of a segment beginning with a temporary label, in the pixel version, cannot exist in the segment version. Then, it should quite always outperform pixel-based techniques. Let us call LSL (Light Speed Labeling) the algorithm described in the next section that we advocate for in the segment-based category.

2 Light Speed Labeling

Like other modern segment-based algorithm, LSL focuses on architecture-algorithm adequacy. The problem of quickly finding out the segment adjacency is reformulated by introducing a new *line-relative* labeling that helps limiting conditional statements. That is all the more crucial as these statements are responsible for pipeline stalls. The line-relative labeling is combined with a Selkow's algorithm to make LSL the most data independent as possible. *Run length encoding* is extensively used at each step of the algorithm. Last but not least, a peculiar attention was paid to *segment-bound* data structures and their implementation to minimize cache misses (section 3).

Let define the following notations:

- er, a relative label,
- ea, an absolute label,
- a, an ancestor label
- X, a binary image of size $h \times w$, X_i the current line of X, and X_{i-1} the previous line.
- EA, an image of size $h \times w$ of absolute labels ea before equivalence resolution
- L, an image of size $h \times w$ of absolute labels ea after equivalences resolution
- ER_i, an associative table of size w holding the relative labels er associated to X_i
- ner, the number of segments of ER_i – black + white –.
- RLC_i, a table holding the run length coding of segments of the line X_i, RLC_{i-1} is the similar memorization of the previous line.
- ERA_i, an associative table holding the association between er and ea: $ea = ERA_i[er]$
- EQ, the table holding the equivalence classes, before transitive closure
- A, the table of equivalence classes after their resolution
- RLC, a 2D table of size $h \times 2w$ holding all segments of every line, used along LSL evolutions
- LEA, 2D list of absolute labels of every line, used in LSL evolutions
- C, the dual of A

The ERA_i table associates the relative and absolute labels on a line. The table is filled up during the top-down label propagation. The relative label is associated with the minimum value of all absolute labels of the intersected segments in the previous line. Of course, the algorithm, is "feed-forward". So, the local minimum is propagated from first to last equivalence. The local minimum equals the global one in the end. Its propagation is actually secured in T, as follows: the local minimum is stored, and from closest to closest, the latest absolute label ea has the global minimum a for its ancestor.

The LSL algorithm is designed to fit RISC processor architectures: memory caches and pipeline execution. LSL accounts for the pipeline by minimizing the number of tests and comparisons performed to detect segments and to find the segments adjacency out. Classically a test makes pipeline

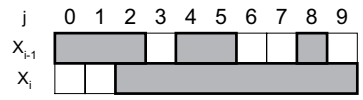

Fig. 7 Lines X_i, X_{i-1}

to stall as the processor needs to know the result of the currently executed instruction before launching the next one. The deeper the pipeline the bigger the impact on the performance. LSL is not a 2-pass algorithm but a 3-pass one. It introduces a pre-pass that performs a line relative labeling devoted to speedup the next passes. Again, the main drawback of segment-based algorithms is they behave like a fusion sort, but with a more complex automaton as segments have length unlike points do . Let us underline that LSL can directly find out the number of adjacent segments and their labels, without performing complex adjacency tests. LSL is composed of five steps:

- **step#1**: first labeling (relative segment labeling)
- **step#2**: equivalence building
- **step#3**: second labeling (first absolute labeling)
- **step#4**: equivalence resolution
- **step#5**: third labeling (final labeling)

Four versions of LSL were written called STD, RLC, XRLC, RLE. The differences come from the sub-algorithms used for the five steps (Tab. 1). The first step could be data independent or conditional to pixel values. Steps 2, 3 and 5 can be either pixel-based or segment-based. And in the RLE version, step#3 can be skipped. The specifications of the 4 versions are summarized in the following chart:

step / version	STD	RLC	XRLC	RLE
step#1	indep	cond	cond	cond
step#2	pixel	segment	segment	segment
step#3	pixel	pixel	segment	$\emptyset$
step#5	pixel	pixel	segment	segment

Table 1 LSL versions specifications

The 2 basic versions called STD and RLC were primarily designed with a 4 and 8-connected neighborhood. Then two optimizations of RLC called XRLC and RLE were developped and a final optimization called *zero-offset* (Z for short) was added. The total number of versions is then sixteen: $\{$STD, RLC, XRL, RLE$\} \times \{$4-C, 8-C$\} \times \{\emptyset$, Z$\}$. The differences are:

- the STD version is dedicated to DSPs where conditionnal instructions can lengthen the execution time, so the scan process, first absolute labeling, is unconditional. For sake of building a reference version to compare to other optimized version, the first labeling is also point by point.
- the RLC version uses the same memory allocations, but with a segment labeling and a conditional scan:

– the XRLC version has a full table of RLC segments to perform the final labeling by segment.
– the RLE version is based on the XRLC-version but does not perform the second labeling. Instead of updating the image of relative labels (step#2) ER, the absolute labels ea are stored in a list LEA. After the equivalence resolution, theses labels are used to create the image, like a RLE decompression with RLC and LEA tables.

Let us underline that the distinction between RLC and RLE made in this paper is specific to our problem of architecture fitting: RLE refers here more to the meaning in compression. Actually, run length coding is used at every step in RLE.

The figure 8 represents the synoptics of STD and RLC versions of LSL.

The five steps are fully explicited in the next section and illustrated through the labeling of figure 7. Given an image X of size $h \times w$, the figure focuses on 2 lines: X_i and X_{i-1}, the current and previous lines.

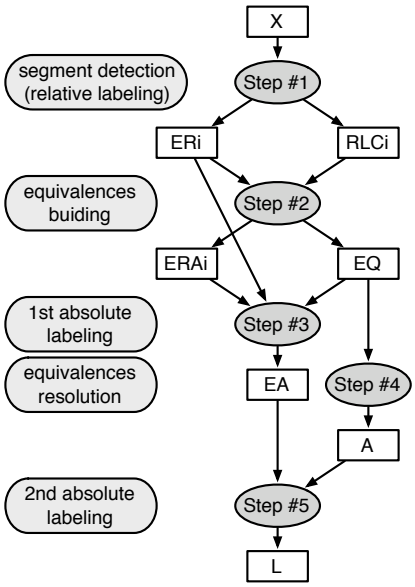

Fig. 8 LSL synoptics of STD & RLC versions

However, the next section is devoted to the sole STD and RLC algorithms. For sake of clarity, the XRLC, RLE versions and Z optimizations will be described in the section after dedicated to LSL optimizations.

2.1 Relative segment labeling: **step#1**

Step#1 performs a relative labeling of each line. For each line X_i the ER_i table holds the associated relative label er of each segment. *relative* refers to that a same numbering (restarting from zero) is performed for every line. As segments are separated by slices of background pixels, an efficient numbering trick consists in assigning odd numbers to segments and even numbers to background (Fig. 9). While labeling segments, their run length code (begin and end $[j_0, j_1]$ of each segment) is also stored into the RLC_i table.

Algorithm 12: LSL segment detection STD

```
   Input: X_i a binary line of width w
   Result: ER_i, RLC_i and ner
 1 x_1 ← 0 previous value of X
 2 f ← 0 front detection
 3 b ← 0 right border compensation
 4 er ← 0
 5 for j = 0 to w − 1 do
 6     x_0 ← X_i[j]
 7     f ← x_0 ⊕ x_1
 8     RLC_i[e_r] ← j − b
 9     b ← b ⊕ f
10     er ← er + f
11     ER_i[j] ← er
12     x_1 ← x_0
13 x_0 ← 0
14 f ← x_0 ⊕ x_1
15 RLC_i[er] ← w − b
16 er ← er + f
17 ner ← er
18 return ner
```

In algorithm 12, f represents the front detection of a segment and is computed with a XOR (noted $\oplus$) while b performs a correction of the end of segment. Indeed this end of segment is detected one pixel after the real segment end. Memory accesses are also optimized through the introduction of two registers: $x_0 \leftarrow X_i[j]$ the current pixel and $x_1 \leftarrow X_i[j-1]$ the previous one. Such a register rotation (line 12), saves one memory access on four, that is 25%. Note that this algorithm is fully **data independent**. The result of its execution is given figure 9. The epilog after the loop is to tackle the border problem and to process the last point without reading a point beyond the end of the line. One can notice that the cell $RLC_i[er]$ is written more than once, if the current segment is longer than one pixel. This can be avoided with the algorithm 13 called segment labeling RLC, as it prevents multiple accesses to the RLC_i table. Algorithm 13 is not data independent but it is hoped to be faster.

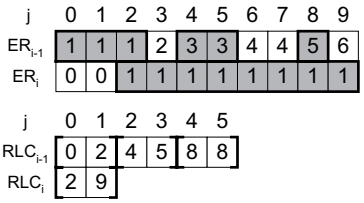

Fig. 9 tables ER_{i-1}, ER_i, RLC_{i-1} and RLC_i

Such kind of a numbering (Alg. 12 line 10) is known in the field of parallel computing as refering to the "scan"

Algorithm 13: LSL segment detection RLC

```
   Input: X_i a binary line of width w
   Result: ER_i, RLC_i and ner
1  x_1 ← 0 previous value of X
2  f ← 0 front detection
3  b ← 0 right border compensation
4  er ← 0
5  for j = 0 to w − 1 do
6      x_0 ← X_i[j]
7      f ← x_0 ⊕ x_1
8      if f ≠ 0 then
9          RLC_i[er] ← j − b
10         b ← b ⊕ 1
11         er ← er + 1
12     ER_i[j] ← er
13     x_1 ← x_0
14 x_0 ← 0
15 f ← x_0 ⊕ x_1
16 RLC_i[er] ← w − b
17 er ← er + f
18 ner ← er
19 return ner
```

concept (4). Operations of that type are defined more fundamentally as follows:

- given an associative operator $\diamond$ and a vector $v(x)$, $0 \leq x \leq n_N$,
- the $\diamond - scan$ of v produces a vector $w = \diamond - scan(v)$ such that: $w(x) = v(0) \diamond v(1) \diamond ... \diamond v(x_N)$

Then with operators $+$ and $\oplus$, it comes:

$$ER_i[j] = \Sigma_{k=1}^{k=j} X_i[k-1] \oplus X_i[k] \quad (2)$$

ner is equal to the number of odd and even segments by construction. So the odd segment er is the $er/2$-th odd segment of the line and its boundaries $[j_0, j_1]$ are stored into $RLC_i[er-1]$ and $RLC_i[er]$ respectively. In our example, the boundaries of the segment $er = 1$ are $RLC_i[0] = 0$ and $RLC_i[1] - 1 = 10 - 1 = 9$.

2.2 Equivalence construction: **step#2**

Step #2 is the equivalence construction (Algo. 14).
For each segment er, its boundaries $[j_0, j_1]$ are read from RLC_i (and are modified in the case of 8-connected labeling) to direclty obtain the relative labels of every adjacent segment in the previous line: er_0 is the label of the first segment and er_1 the label of the last segment. As background slices are labeled with even numbers, a correction, based on parity check is applied to er_0 and er_1 (lines 11,12). The number of adjacent segments is trivially $(er_1 - er_0)/2 + 1$. If there is an adjacency, the absolute label ea of the first segment is read from the associative table ERA_{i-1} that holds the bijection between relative and absolute labels. The ancestor a, that is the smallest label of the equivalence class is initialized with the label that is equivalent to ea.

The loop consists of extracting the absolute label ea_k and the ancestor a_k of each adjacent segment then propagating

Algorithm 14: LSL equivalence construction

```
   Input: ER_{i−1}, RLC_i, EQ, ERA_{i−1}, ERA_i, ner
   Result: nea the current number of absolute labels, update of
           EQ and ERAi
1  for er = 1 to ner step 2 do
2      j_0 ← RLC_i[er − 1]
3      j_1 ← RLC_i[er]
4      [check extension in case of 8-connect algorithm]
5      if j_0 > 0 then j_0 ← j_0 − 1
6      if j_1 < n − 1 then j_1 ← j_1 + 1
7      e_r0 ← ER_{i−1}[j_0]
8      e_r1 ← ER_{i−1}[j_1]
9      [check label parity: segments are odd]
10     if e_r0 is even then e_r0 ← e_r0 + 1
11     if e_r1 is even then e_r1 ← e_r1 − 1
12     if e_r1 ≥ e_r0 then
13         e_a ← ERA_{i−1}[e_r0]
14         a ← EQ[e_a]
15         for e_rk = e_r0 + 2 to e_r1 do
16             ea_k ← ERA_{i−1}[er_k]
17             a_k ← EQ[ea_k]
18             [min extraction and propagation]
19             if a < a_k then
20                 EQ[ea_k] ← a
21             else
22                 a ← a_k
23                 EQ[ea] ← a
24                 ea ← ea_k
25         ERA_i[er] ← a the global min
26     else
27         [new label]
28         nea ← nea + 1
29         ERAi[er] ← nea
```

the minimum ancestor to every label. At the end of the loop, a is equal to the global minimum of all ancestors a_k. That value becomes the new absolute label of segment er and is memorized into the ERA_i table. In the case of no adjacent label, a new label is created and the total number of absolute labels nea is incremented.

Note that switching from the 8-connected version (Algo. 14) to the 4-connected one is straightforward: it is enough to remove the *diagonal lookups* of lines 5 and 6. That makes the only difference between the two versions !

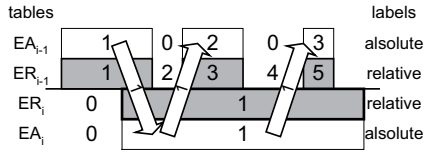

Fig. 10 Propagation of absolute labels through to relative labels

Let us stress upon that, in this segment case, the Selkow's algorithm is even simpler and deterministically so with only one access to the equivalence table (Algo. 14, lines 15 and 18). For instance any pixel configuration as in figure 5 still requires a single access. Thus, for a segment-based algo-

rithm Selkow's has the smallest possible complexity. In particular, it is less complex than the Union-find algorithm which complexity grows theoretically like the inverse Ackermann's function (Sec. 4.3) as there is no growth at all.

2.3 First absolute labeling: **step#3**

Algorithm 15: LSL segment first absolute labeling

```
1 for i = 0 to h − 1 do
2     for j = 0 to w − 1 do
3         EA_i[j] ← ERA_i[ER_i[j]]
```

Step#3 consists in replacing the relative label of every segment by its absolute label (Algo. 15). This step is straightforward: ERA_i can be interpreted as a *Look Up Table* to be applied to ER_i to create EA_i.

2.4 Equivalence resolution: **step#4**

Step #4 is the resolution of the equivalence classes. The Selkow's algorithm is preferred to Rosenfeld's for building equivalences as it is more *stable* (*ie* runtime predictable). Yet, both resort to the same algorithm in computing the transitive closures (Algo. 6) .The EQ table is solved and packed into the associative table of ancestors A.

2.5 Second absolute labeling: **step#5**

Algorithm 16: LSL second absolute labeling

```
1 for i = 0 to h − 1 do
2     for j = 0 to w − 1 do
3         EA_i[j] ← A[EA_i[j]]
```

Step #5 is identical to step#3: every absolute label ea is replaced by its ancestor a (Algo. 16). This step is equivalent to the second Rosenfeld's scan (Algo. 7).

3 Algorithmic and Architectural optimizations of LSL

We present in this section, some algorithm transformations and software optimizations that will lower the execution time. We start with algorithmic transformations, as they have the biggest impact on the performance.

3.1 Algorithmic optimization: XRLC & RLE

Like every segment-based CCL, the 8-connected version of the LSL is very close to the 4-connected one: just two additions to add in reading the pixel (Algo. 14 lines 6-7).

As the relative segment labeling (Algo. 12 & Algo. 13), the algorithms 15 and 16 can be modified to use an RLC approach. Since every pixel of a segment er have the same absolute label ea, that label can be read only once and applied to the entire segment. That modification implies that a RLC table be created to hold the *boundaries* of every segment of every line. It implies a memory allocation of size $height \times ner_{max}$, with ner_{max} the maximum number of segments per line. We have $ner_{max} < width/2$ (happens when a line is fully dashed). This extension of RLC to the whole algorithm is called $XRLC$. This modification speeds up the filling of EA.

The next evolution is to reduce the amount of memory accesses again, by performing a compression based on *run length coding* each segment. In that case the image EA is no more written. The absolute labels ea of every segment are stored in a list LEA_i. There is one list per line. The image EA can be reconstructed (for human visual check) by first applying A to LEA and then *uncompressing* RLC with associated values of LEA.

We can gain even further: as said in introduction, a labeling algorithm is an intermediate algorithm that is used after a binary segmentation and before some feature computation. Usually such a feature computation runs on the labeled image, i.e. *after* the global end of a labeling algorithm: after the transitive closure and after the second pass of re-labeling. But if the features can be computed earlier, then these steps are no more useful, except for humans to visually check the labeling result. That is the goal of the RLE version. A new table called LEA is filled up *on the fly* at step#2 (Fig. 11) and holds the *list* of every absolute labels ea. Then, RLC and LEA tables are enough to create the final image of labels L without accessing the image of absolute labels EA.

Yet, to be efficient, the RLE version of LSL should benefit from a smart memory implementation.

3.2 LSL and Memory management

Proper memory management needs to be secured. Indeed, evolutions and adding optimizations means allocating more and more *oversized* structures in memory. Since the final number of labels is unknown at the beginning, the waste is mandatory to avoid sparse addressing responsible for cache misses and big penalties to CPU execution time. First, 2D associative tables (RLC, LEA and C) are allocated with *Numerical Recipes in C* `matrix` routines (19) for 16-bit and 32-bit numbers. Being based on *offset addressing*, these routines are easily customizable. A `matrix` is a 2D array with a 1D array of pointers holding the start of each line (Fig. 12, left part). Once a line of width n is filled with m elements ($m \leq n$), the pointer to the next line is modified to

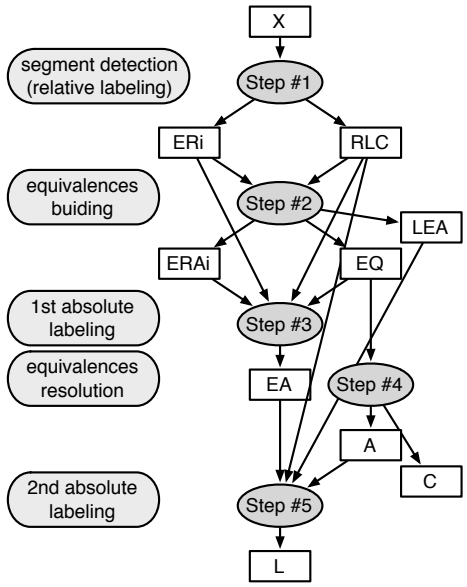

Fig. 11 LSL synoptics of XRLC & RLE versions

point to the first free cell of the current line. To begin with, *start of line* pointers are equidistant, that is $p_{i-1} = p_i + n$ (using pointer arithmetic in C). From there it just remains to set p_{i+1} to $p_i + m$, and so on, for every line i. This ensures storing contiguous data into memory and maximizing cache hits.

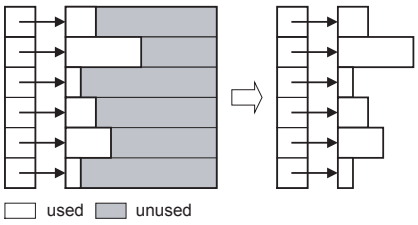

Fig. 12 LSL memory management for large tables

Other feature computations by other developpers (see section 3.3) may appear necessary in the future. Aiming at efficient add on, an associative table C is also constructed at step 4. C is an associative table, *dual* of A: table A is implementing a Union-Find structure, where $A[ea]$ points to a, the common ancestor of the equivalence class. Given an ancestor a, $C[a]$ holds the set of absolute labels ea that belong to that equivalence class. The table C is the union of all C_a for every a. So finding the different labels ea composing the class a, one needs only to read C_a, where labels ea are already sorted. Such a kind of access enforces the memory cache motto that is **optimizing spatial and temporal localities**. Computing extra features is then faster than computing the features of all classes together. It avoids sparse footprints into memory and it avoids to modify the existing algorithms.

3.3 LSL and feature computation

There are mainly two kinds of features:

1. geometrical features, like bounding rectangles $[i_0, i_1] \times [j_0, j_1]$,
2. radiometric features like statistical moments e.g. the zero-order moment S represents the surface of the component and the first-order moments S_X and S_Y provide the centroid $(x_G, y_G) = (S_X/S, S_X/S)$,

These features can be computed at the end of the CCL when the absolute label of each segment is known, or on the fly if integrated to the algorithm 14. The parameter i_0 is initialized at the creation of a new label. Likewise i_1 is initialized and then overwritten by the current i, as long as ea exists. The values of j_0 and j_1 are also initialized at the creation and updated with the min and max values for the upcoming segment ea.

The statistic moments can also benefit from the RLC coding. Let be s, s_x and s_y the zero-order and one-order moments computed for a segment. S, S_X and S_Y are the accumulation of them for every segment of absolute label ea. For a segment, the following formula hold:

$$s = j_1 - j_0 + 1$$
$$s_y = i \times s$$
$$s_x = \varphi_1(j1_1) - \varphi_1(j_0 - 1), \text{ with } \varphi_1(n) = \frac{n(n+1)}{2}$$

Higher order moments can be also computed fastly with Bernoulli polynomials $\varphi_p(n)$:

$$\varphi_p(n) = \sum_{x=1}^{x=n} x^p$$

These formulae are all the more interesting as the computation of statistical moments becomes independent of the segment width: here again the procedure is *data independent*.

As previously said, these computations can be completed *on the fly* or *after* labeling thanks to RLC, LEA, and C tables. But in both cases the time consuming labeling steps of LSL (step#3 and step#5) are no more required. The Benchmark (Tab. 6) will show that LSL with feature computation but without steps #3 and #5 is running at quite the same speed as LSL without feature computation but with these steps: theRLE version makes feature computation a *free lunch*.

3.4 Zero-Offset addressing

The last optimization to be done is *zero-offset* addressing. It could seem unsignificant but benchmarks have shown a speedup of 5%. Instead of storing j_0 and j_1 – the actual boundaries of a segment – that also requires the register b to correct j_1, the value $j_1 + 1$ will be stored into RLC. This

Algorithm 17: Selkow segment detection STDZ

Input: X_i a binary line of width w
Result: ner the number of relative labels on the line X
1 $x_1 \leftarrow 0$ previous value of X
2 $f \leftarrow 0$ front detection
3 $er \leftarrow 0$
4 **for** $j = 0$ **to** $w - 1$ **do**
5 $\quad x_0 \leftarrow X_i[j]$
6 $\quad f \leftarrow x_0 \oplus x_1$
7 $\quad RLC_i[e_r] \leftarrow j$
8 $\quad er \leftarrow er + f$
9 $\quad ER_i[j] \leftarrow er$
10 $\quad x_1 \leftarrow x_0$
11 **if** $x1 \neq 0$ **then** $RLC_i[er] \leftarrow w$
12 $er \leftarrow er + x_1$
13 $ner \leftarrow er$
14 **return** ner

leads to an even smaller and faster algorithm for relative labeling (Algo. 17).

The other algorithms should be also slightly modified. During the equivalences building (Algo. 14) the right boundary stored is $j_1 + 1$. Line 4 should be replaced by $j_1 \leftarrow RLC_i[er] - 1$. s_x that was previously equal to $s_x = (j_1(j_1 + 1) - j_0(j_0 - 1))/2$ should be replaced by $s_x = (j_1(j_1 - 1) - j_0(j_0 - 1))/2$. The complexity of s_x remains unchanged while the line-relative labeling complexity drops (Algo. 17). That is an optimization *without counterpart*.

3.5 Algorithmic complexity

In this section we tackle both the memory footprints and the number of memory accesses, other key instructions as comparisons may be procedure dependent.

The Rosenfeld's algorithm requires 2 images (1 for data, 1 for labels) of size $h \times w$ and 1 equivalence table of size n to store the labels during the first pass and the ancestors at end. The LSL algorithm, in its STD version, requires 5 extra tables to store ER_i, ER_{i-1}, RLC_i, ERA_i, ERA_{i-1}. As the maximum number of *relative* labels is bounded by $w/2$ in the case of black and white pixel alternance, ER and ERA tables hold both odd and even labels so their size is bounded by w. RLC_i table deals with even labels while holding 2 parameters per label whence the w upper bound again. For the RLE version, there are 2 additional 2D tables with height h: RLC and LEA then bounded by $h \times w$. The table 2 sums up the algorithm memory requirements. Note that LSL_{STD} is very close to Rosenfeld's in term of memory occupation while LSL_{RLE} is twice as complex. In the worst case, the 4-connected chessboard, n reaches $h/2 \times w$ and LSL_{RLE} requires $\times 1.6$ the Rosenfeld's memory amount. Eventually the three algorithms can be embedded the same way as they have roughly the same memory footprints.

As for memory access, LSL completes 3 image scans (step#1, #3, #5) including vizualisation. An important benchmark result turns out to be that the 3-pass LSL_{STD} is faster than the 2-pass Rosenfeld$_{DT}$. Obviously, memory accesses do not have the same duration. Due to the cache size, step#1 and step#2 likely access data fitting in the cache and remaining in it at step#3. At LSL's step#5 and Rosenfeld's second labeling, data would not be in the cache anymore. Reloading typically costs a dozen CPU cycles from the level 2 cache and more than two hundred ones from the external memory. We think an accurate further analytical model is not doable and choose to rely on a rationally-constructed and large-enough benchmark for further analysis.

algorithm	memory footprint	worst case
Rosenfeld	$2hw + n$	$2.5hw$
LSL_{STD}	$2hw + n + 5w$	$2.5hw + 5w$
LSL_{RLE}	$2hw + n + 4w + 3hw/2$	$4hw + 5w$

Table 2 Procedure's memory footprints

The data-dependent steps are Union-Find for Rosenfeld and step#2 (equivalence building) for LSL. Behaving like a merging sort, the LSL step#2 complexity is proportional to the number of segments that is $w/2$ in the worst case. Comparatively, the complexity of the Union-Find in Rosenfeld's is proportional to the height of the equivalence tree again bounded by the inverse Ackermann function.

Finally in terms of worst case, considering that a segment algorithm has twice the complexity of a pixel one since it processes two data – the segment boundaries – instead of one – the pixel, both would encounter the same worst case as the maximum number of segments is half the line width. While being not as universal, plausible benchmarks provide more precise results, of course, to be further interpreted. That we endeavor to in the next section.

4 Benchmarks: Algorithms Evaluation and Results Analysis

4.1 Benchmark data

CCL algorithms are *data dependent* and benchmarking such algorithms is not obvious. We propose a four stages process depending on the growing data intricacy. The first step is to evaluate the algorithms in the *maximum stress case*, that is intuitively here represented by totally unstructured data – random images – especially hard on segment algorithms. The second stage is then to test quasi-structured data. In our case previous images are filtered with morphological operators, to remove stand-alone pixels and to cluster others. The third stage is to test highly structured data (homothety) where the number of labels and ancestors are exactly the same for all algorithms (pixel or segment CCL, 4 or 8 connected CCL).

Let us underline that stages two and three allow to evaluate figures of merit vs. parameters – e.g. number of dilations, square size – that refer to the average size of regions and thus to the expected number of intermediates labels.

Finally, real images are tested that involve many labels to be representative enough. In that category, we include images commonly used for benchmarking CCL algorithms like Sidba, Brodatz and Waterloo databases. In view of result interpretation and benchmark plausibility-check, the table 4 displays the number of labels per image type (Sidba, Brodatz, Waterloo, percolation, 3×3 dilation, OCR, and cadastre).

Indeed, it is known that in applications requiring *on the fly computations* the proof of ultimate validity would be the worst-case optimality. In the absence of a suitable generic image model to support the design of the most stressing data, the present paper first relies on artificial sample images chosen as explained above to be likely stressing on the CCL procedures in a controlled manner. Then selected real images are tackled.

Even though real images would be considered with various versions of them from adding noise in different manners and binarizing optimally (20), such testing process would not allow for data resizing. Conversely, the technique of generating images in random manners as proposed in this paper allows image sizes as large as necessary. And from the point of view of the intrinsic data complexity, it can be assessed through table 4 that the latter images are *a priori* more difficult by a factor from $\times 5$ to $\times 20$ depending on the number of concavities or stairs and then of labels.

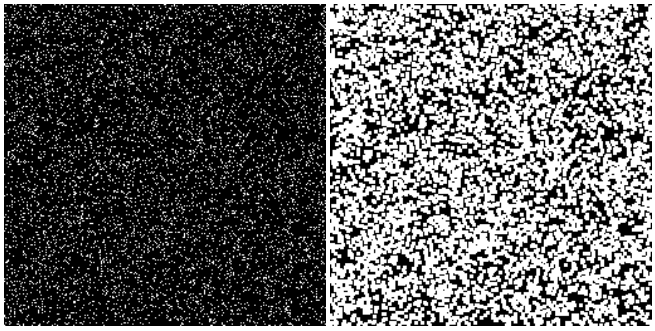

Fig. 13 images of percolation with a threshold at 0.900: raw (left) and after a 3×3 dilation (right)

Bello,w the four image test-benches are precised:

- percolation (Fig. 13, left part): for image size $n \times n$ with $n \in \{256, 512, 1024, 2048\}$, 1000 images are randomly generated with a density varying from 0 to 1000 by steps of $1/1000$.
- mathematical morphology (Fig: 13, right part) to get slightly structured images, a morphological operator is applied to previous percolation images: erosion, dilation, opening and closing with a structuring element of size 3×3 or 5×5.
- structured data & *homothety*: images are paved with squares of size k, $k \in \{4, 8, 16, 32, 64, 128\}$. For a given square size k, a set of homothetical images are generated according to a scale factor λ ($\lambda \in 1, 2, 3, 4, 5$) so keeping the number of labels constant. For instance, the image size is $512 \cdot 2^{\lambda} \times 512 \cdot 2^{\lambda}$ while the squares are $k \cdot 2^{\lambda} \times k \cdot 2^{\lambda}$ (Fig. 14).
- *real life* & structured images: we picked images first in Optical Character Recognition (OCR) and automatic cadastre analysis for an intuitive, yet complex extension of percolation and square images. Then we picked images from usual databases in this field that were confirmed to be simpler (Tab. 4).

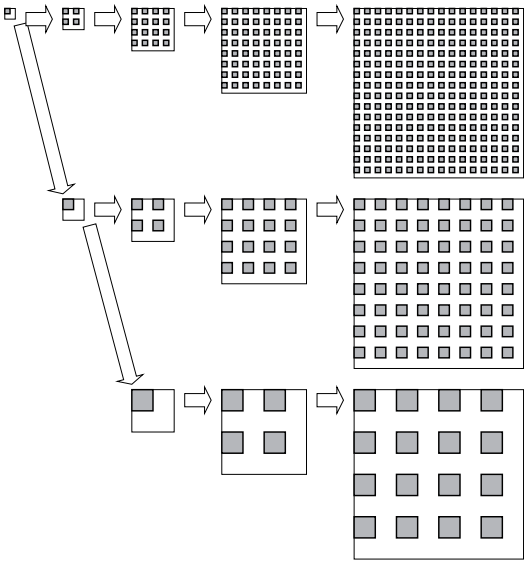

Fig. 14 homothetical images famillies

The percolation images serve evaluating the behavior of labeling algorithms with *random* images but also the impact of the density on the number of generated labels vs. the execution time. Morphological images are used to test images that are a bit structured, but less than natural images. Homotheties are used to evaluate the impact of the relative ratio object / image sizes. Such data enable to separate between the number of components and the image size when execution time varies. OCR images are interesting as OCR is an important application that requires realtime execution (Post Offices for example). OCR images can contain a huge number of regions with small size. Cadastral images are harder to label: some regions are even smaller and split into sub regions (due to black and white hatching) and they are rounded by very large regions (street, buildings). For OCR, the Universal Human Right Declaration was processed: the Declaration was written with character case from 8 to 12 and converted into images with resolutions from 72 dpi to 300 dpi. Then, to push the algorithms, the resulting pages are pasted together to get very large images. The size of the biggest image is 2333×12163 pixels. Images data sets are available at (14).

4.2 Benchmark metrics

Table 3 describes the two families of processors used for benchmark: the Motorola PowerPC 7447 (aka G4), the IBM PowerPC 970MP (aka G5), the Intel Conroe and Intel Penryn. On multi-core processors, only one core was triggered.

arch	Freq	L2 cache size	techno	compiler
G4 (PPC477x)	1 GHz	256 KB	130nm	gcc 4.0
G5 (PPC970)	2.3 GHz	1024 KB	90nm	gcc 4.0
Conroe (T7700)	2.4 GHz	4096 KB	65nm	icc 10.1
Penryn (Q9550)	2.8 GHz	6144 KB	45nm	icc 10.1

Table 3 processors spec and compilers used

Comparing architectures from a quantitative point of view, we choose to rely on the *cpp* (*Cycle Per Point*). It is an architectural metric to estimate the adequacy of an algorithm to an architecture (12). On constant frequency processors, the *cpp* is the normalized execution time: $cpp = t \times F/n^2$. t is the execution time, F the processor frequency and n^2 the number of pixels to process. On variable frequency processors, *cpp* is computed from 64-bit hardware cycle counters available on Intel and PowerPC. They are readable from operating systems in use. As the execution time is normalized by the processor frequency and the image size, results from a given algorithm running on a given architecture can be compared to other ones.

Three algorithms and six configurations are evaluated in this paper. First, pixel-based algorithms are compared together, with and without optimization (Decision Tree, Path Compression) to evaluate the impact of the systematic Selkow's double access versus the classical Union Find *Ackermann*-optimal access. Second, the fastest optimal pixel algorithm (28) is chalenged by LSL. Two versions of LSL are involved: STD, the most systematic one and RLE, the more data dependent one. For each benchmark, we provide the *cpp* but also the standard deviation (sd) that is a fair indicator of the global behavior of each algorithm: the smaller sd, the more runtime predictable. For structured data, two *cpp* are provided – with and without feature computation – to assess the algorithm behavior in a real application.

Let define the following procedures *short name*:

- R_{UF}: Rosenfeld procedure based on classical Union-Find Structure
- R_{PC}: Rosenfeld procedure based on classical Union-Find Structure with Path Compression
- R_{DT}: Rosenfeld procedure using Path Compression and Decision Tree (28)
- S_{DT}: Selkow procedure with Decision Tree
- LSL_{STD}: LSL procedure with systematic computations (STD) (L_{STD} for short in the table)
- LSL_{RLE}: LSL procedure with RLE compression (L_{RLE} for short in the table),

All procedures were written strictly in the same manner to enforce the same level of software optimization. For instance, the implementation is based on *pointer to function* calls: the same loop (with `i` the line number) calls the functions associated to the algorithms and their versions. The only drawback of such coding fashion is that it prevents from *Inter-Procedural Optimization* for compiler option. The granularity of the pointer to function is at line level. Unlike, with pixel level granularity, its impact on performance is then minor and all procedures are impacted the same way. Not to forget that it makes an efficient way in C towards extensible system of benchmark to handle many algorithms ($\{$ 8-bit, 16-bit, 32-bit $\} \times \{$ 4-connected, 8-connected $\} \times \{$ algorithms and versions$\}$). For tests to further guaranty equitable benchmark, the website (14) gives access to binary executables and all the images used. That enables other programmers to cross-check.

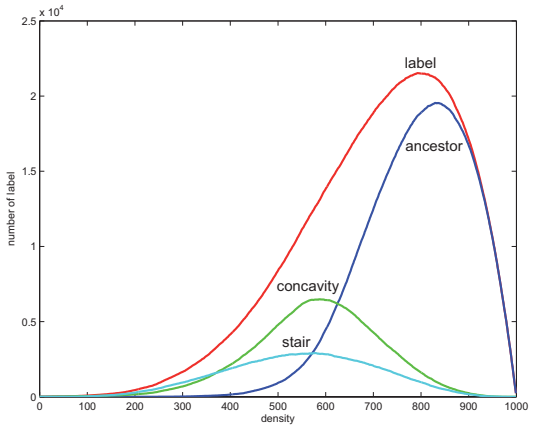

Fig. 15 complexity for random percolation

4.3 Benchmark results and analysis

The trend of the results is similar with the four processors (Tab. 6 and 5), so for conciseness the following ratios and speedups refer only to the Penryn processor as being the state-of-the-art off the shelf processor.

Considering the optimizations of Rosenfeld's algorithm, the Path Compression (PC) is not efficient whatever the processor or the benchmark (columns R_{UD} and R_{PC} of table 5). This significative result makes a counter-example: the procedure execution time can not be directly related to the algorithm complexity. As said in section 1.3.1, PC is a general Graph Theory optimization while DT accounts for the local pixel topology. As a consequence, R_{DT} is up to $\times 1.33$ faster than R_{PC} (columns R_{DT} of table 5).

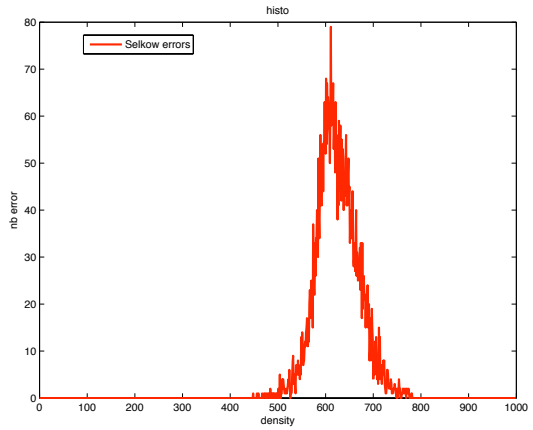

Fig. 16 distribution of Selkow'algorithm errors for 400.000 images. Error rate is 1.86 %. Histogram is correlated with the number of concavity - Fig. 15

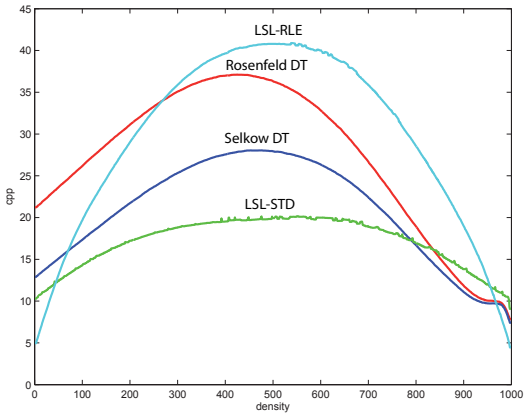

Fig. 17 Penryn *cpp* for random percolation

data base	label	ancestor	concavity	stair	depth
Average values					
SIDBA	1400	624	354	422	7.0
Brodatz	424	174	130	119	4.6
Waterloo	2164	1170	594	400	15.4
random perco	8817	5578	2042	1181	116.4
dilation	711	222	156	332	4.8
OCR	34311	9428	5562	19321	32.4
cadastre	8377	3036	538	4803	17.9
Max values					
SIDBA	4435	2083	1199	1450	43
Brodatz	1262	820	394	337	35
Waterloo	6481	3808	1973	1239	30
random perco	21530	19552	6487	2911	3533
dilation	5306	3545	1584	2212	238
OCR	-	-	-	-	54
cadastre	-	-	-	-	33

Table 4 databases spec and complexity: remind that nb of label = nb of ancestors + nb concavity (in segment mode) and + stairs (in pixel mode); their repartition vs. the depth matters.

arch	R_{UF}	R_{PC}	R_{DT}	S_{DT}	L_{STD}	L_{RLE}
average cpp for random percolation						
G4	69.6	72.1	48.5	49.2	46.8	42.0
G5	56.7	58.4	35.1	33.8	25.9	34.8
Conroe	43.4	45.0	32.4	31.6	23.6	30.8
Penryn	36.2	42.1	27.2	20.8	**17.1**	29.7
standard deviation in cpp for random percolation						
G4	19.1	19.3	6.7	6.2	4.2	8.4
G5	22.0	23.3	8.9	8.1	4.7	10.8
Conroe	14.1	14.5	9.6	9.5	3.5	11.2
Penryn	14.3	18.0	8.8	6.1	**2.9**	10.5
average cpp for dilation 3×3						
G4	52.4	54.7	42.3	44.9	37.7	24.3
G5	36.8	35.6	25.9	25.9	16.8	15.4
Conroe	39.2	41.3	26.5	28.7	16.2	9.8
Penryn	**20.9**	20.3	22.1	13.9	11.1	**9.1**
standard deviation in cpp for dilation 3×3						
G4	9.2	9.1	2.4	2.5	1.5	3.5
G5	8.3	9.4	2.8	2.7	1.8	5.1
Conroe	5.5	5.6	3.1	3.4	1.3	5.7
Penryn	7.3	6.7	2.6	**2.1**	1.0	**6.2**

Table 5 percolation & dilation 3×3 results in *cpp*

Moreover, for all benchmarks (Tab. 5 and 6), S_{DT} is always as fast or faster than R_{DT} and has always a smaller standard deviation. In other words, it is more runtime predictable. That enlightens the key part of the UF structure management (even with PC and/or DT) in the computing cost. Then the usual claim that the UF tree depth grows like the inverse of Ackerman's function and thus makes PC efficient turns out to be inaccurate. So the Selkow's algorithm should be preferred to Rosenfeld and Union-Find algorithms especially in applications where *video frame rate* is mandatory. As announced in sections 1.3.1 and 1.3.3, the Selkow's algorithm is faster but probabilistic: a supplementary run on a 400.000 percolation image benchmark – the most stressing type in terms of concavities – brings out a failure rate of 1.86 %. The latter images are parameterized by the density of white pixels and the histogram (Fig. 16) indicates the correlation of its peak value with the maximum number of concavities (Fig. 15).

For random images, again the most stressing cases for a segment-based CCL, LSL_{STD} is faster than all pixel-based procedures (Fig. 17): $\times 1.6$ faster than R_{DT} with a $\times 3$ smaller standard deviation, and $\times 1.2$ faster than S_{DT} with a $\times 3$ smaller standard deviation.

For structuring elements 3×3 and 5×5, erosion is close to opening and dilation is close to closing. It appears from their sketch that erosion and dilation are well symetric, then we can restrict the study to dilation. For 3×3 dilation images, LSL_{STD} is $\times 2.0$ faster than R_{DT} and its standard deviation drops down to 1.0 that is $\times 2.6$ smaller than R_{DT}. LSL_{RLE} is in average $\times 2.4$ faster than R_{DT}.

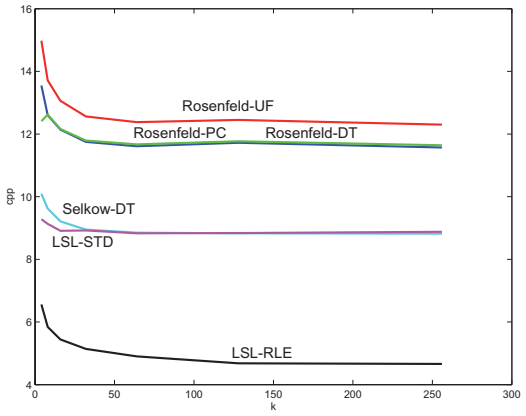

Fig. 18 average *cpp* for homothety, $k \in \{4, 8, 16, 32, 64, 128, 256\}$

arch	R_{UF}	R_{PC}	R_{DT}	S_{DT}	L_{STD}	L_{RLE}
*Homothety **without** Features computation*						
G4	31.7	33.8	29.5	31.2	34.4	22.0
G5	18.0	18.5	16.5	17.1	16.2	13.0
Conroe	18.8	20.5	13.0	13.5	13.7	7.5
Penryn	13.1	12.1	12.0	9.2	9.0	**5.3**
*standard deviation for Homothety **without** Features computation*						
G4	3.6	3.7	1.5	1.5	0.70	1.3
G5	1.4	1.5	0.54	0.52	0.47	1.4
Conroe	1.1	1.0	0.50	0.49	0.48	1.1
Penryn	1.4	1.1	0.82	0.57	**0.38**	0.83
*Homothety **with** Features computation*						
G4	76.4	78.5	74.2	75.9	16.2	16.1
G5	38.1	38.6	36.6	37.2	9.4	8.8
Conroe	30.9	32.6	25.1	25.6	7.2	5.1
Penryn	21.7	21.6	21.5	19.0	6.0	**4.5**
*OCR **with** Features computation*						
G4	71.3	73.9	68.3	69.7	16.1	17.1
G5	35.7	36.4	32.4	32.3	17.4	14.3
Conroe	28.2	29.7	22.8	22.8	7.3	6.2
Penryn	22.1	22.0	19.8	18.4	6.1	**5.1**
*cadastre **with** Features computation*						
G4	116	117.9	106	108	14.9	17.1
G5	50.8	51.3	42.3	42.2	17.2	15.2
Conroe	60.9	63.0	49.4	50.8	7.6	7.9
Penryn	41.5	42.2	42.8	36.2	6.6	**6.8**

Table 6 homothety, OCR and cadastre results in *cpp*

In the peculiar case of homothety, it was necessary for sake of results readability and conciseness to check the stability of execution time vs. square (k) sizes (Fig. 18).

In varying them as above-mentioned and storing the *cpp* in the $k \times \lambda$ matrix of results (λ: scale factor $\simeq$ image size) it appears that the variance of the elements respectively for a given k or a given image size is low (Tab. 6). constant over result subsets and more interestingly independent of the procedure. This constancy allows to display only mean and deviation values global to the whole set of tested images. For homothety, all versions show a decreasing *cpp* but the ratio remains the same when there is no feature computation: LSL_{RLE} is $\times 2.3$ faster than R_{DT}. When there are some feature computation, R_{DT} becomes $\times 1.8$ slower, while, thanks to RLE compression, LSL_{RLE} performance increases to become $\times \mathbf{4.7}$ faster than R_{DT}.

The table 6 figuring the respective execution times of Rosenfeld and LSL procedures show that the variances in both cases range respectively from 0.82 to 1.4 and from 0.38 to 0.83. Being comparable, these values indicate that the impact of the cache is not major, whether data stay in the cache or not the cache predictor succeeds in maintaining the flow. Additionally, would the cache have an influence, the curves (*cpp* vs. k, vs. λ) would present a discontinuity which is not the case (Fig. 18). Thus the main difference in the mean execution times cannot be else than in the amount of unnecessary accesses to the equivalence table and operations to manage pixel/segment adjacency.

For OCR, LSL_{STD} and LSL_{RLE} are respectively $\times 3.3$ and $\times 3.9$ faster than R_{DT}. One can notice too that with feature computation, OCR results are close to dilation results. For very complex images like cadastre, with a huge number of labels and concavities compared to OCR, R_{DT} is much more *data sensitive* than LSL: R_{DT} performance drops down by a factor $\times 2.2$ while LSL_{STD} and LSL_{RLE} respectively drop down by a factor $\times 1.1$ and $\times 1.3$. Thus both LSL versions outperform R_{DT} by a factor greater than $\times \mathbf{6}$.

As a matter of fact, the LSL execution times for this 512×512 images benchmark on a 2.8 GHz Penryn are 1.6 ms for random images, 0.47 ms for OCR and 0.61 for cadastre.

To conclude on the test result analysis, LSL is always faster and more data independent than pixel-base procedure even in the worst case of random images. For unstructured random images LSL_{STD} should be chosen while LSL_{RLE} should run on all other images. More important, if random images can be considered to be *close* to the worst case and homothety images *close* to the best one then OCR/cadastre images should represent the real average case. Under this assumption, LSL execution time is by far closer to the best case than to the worst. For real and complex images, when a component labeling algorithm is considered as a part of a processing chain, that is associated with some feature computation, the speed ratio reaches a level of $\times 4$, proving the importance and the impact of software (cache and pipeline) and algorithmic (*line relative labeling* and RLE compression) optimizations.

Finally, the counter-performance of R_{DT} strengthens the evaluation strategy adopted in this paper (Sec. 4.1 and 3.5): dependable measures with small standard deviation prevail on a complexity model that can not seize the execution time with accuracy.

5 Conclusion and future work

Two algorithms were introduced in this paper. The first one is an evolution of the classical pixel-based Rosenfeld's algo-

rithms where the equivalence construction, commonly based on the Union-Find algorithm, is replaced by a Selkow's algorithm. This algorithm already exists in the French image processing community. When modified with Decision Tree optimization, it is faster and more runtime predictable than the 2007 Wu's *world fastest algorithm* (28).

The second algorithm called *Light Speed Labeling* deals with segments. We focus on RISC architecture specificities to design it. To be efficient a CCL algorithm should optimize pipeline executions by reducing the number of stalls, and should limit the memory footprints and cache misses. We introduce a new *line-relative* labeling that makes the segment adjacency detection more efficient. Combined with Selkow's algorithm, this algorithm has much less conditional statements, whence reducing the number of pipeline *stalls*. As memory management of tables is also a weak point of segment-based algorithms, the implementation of user data structures was also optimized. Sixteen versions of LSL have been designed. Two versions were presented: the first one is the most systematic and data-independent possible and is designed for noisy images (random or pseudo random images with very few structuration) and for systems where predictability is important. The second one is optimized for real images (conditional statements, cache footprints and cache hits).

For the four sets of benchmark, all results point out that LSL is faster (up to $\times 6$) and more runtime predictable (up to $\times 2$) than pixel-based algorithms. As all segment-based algorithms, LSL is optimal with the number of created labels. The implemented memory optimizations make it well suited for embedded and parallel systems where speed and predictability vs. memory are crucial. That point will constitute the next step of our research in the area.

More generally, these results are also interesting as they provide some hints and a new methodology to design data-dependent algorithms that should be implemented on RISC architectures.

Future work will consider parallel versions of LSL and its application to derivate algorithms like hysteresis thresholding, convex hull computation, geodesic reconstruction, black & white labeling with holes filling (3), and more specialized algorithms like level sets (17) and level lines labeling (5) (9).

References

1. P. Adam, B. Burg, B. Zavidovique, Dynamic programming for region based pattern recognition, ICASSP, pp 2075-2078, 1986
2. H. M. Alnuweiri, V.K. Prasanna, Parallel architecture and algorithms for image component labeling, IEEE Transaction on Pattern Analysis and Machine Intelligence, vol 14,10, October 1992.
3. J. Bajon, M. Cattoen, S. D. Kim, A concavity characterization method for digital objects. Signal Processing, 9,3 pp 151-161. 1985.
4. G.E. Blelloch, Vector Models for Data-Parallel Computing. The MIT Press, Cambridge Massachusetts, 1990.
5. F. Guichard, S. Bouchafa, D. Aubert. A change detector based on level sets. International Symposium on Mathematical Morphology ISMM 2000, Palo Alto, pp 321-330, june 2000.
6. F. Chang, C. Chen, A linear-time component-labeling algorithm using contour tracing technique, Computer Vision and Image Understanding, vol 93, pp 206-220, 2004.
7. J.M. Chassery, A. Montanvert, Géometrie discrète en analyse d'image, Traité des Nouvelles technologies, Hermes. ISBN 2-86601-271-2. pp 200-214, 1991.
8. T.H. Cormen, C.E. Leiseirson, R.L. Rivest, C. Stein, Introduction to Algorithms, Charpter #21, pp 498-522, MIT Press, ISBN 0-262-03293-7, 2001.
9. M. Gouiffès, B. Zavidovique, A Color Topographic Map Based on the Dichromatic Reflectance Model, EURASIP Journal on Image and Video Processing Volume 2008, Article ID 824195, 14 pages, doi:10.1155/2008/824195.
10. R.M. Haralick, L.G. Shapiro, Computer and Robot Vision, volume 1, Addison-Wesley ISBN 0-201-56943-4, pp 31-48, 1992.
11. L. He, Y. Chao, K. Suzuki, A run-based two-scan labeling algorithm, ICIAR 2007, LNCS 4633, pp 131-142, 2007.
12. L. Lacassagne, M. Milgram, P. Garda. Motion detection, labeling, data association and tracking in real-time on RISC computer, pp 520-525, ICIAP 1999.
13. L. Lacassagne. Détection de mouvement et suivi d'objets en temps réel, Paris6 University thesis, France, June 2000.
14. Images data base used for benchmarking: `http://www.ief.u-psud.fr/~lacas/Download/LSL/LSL.html`
15. P. Lamaty, D. Demigny, Opérateur matériel d'étiquetage de régions temps reel et flot de données, GRETSI 1999, `http://hdl.handle.net/2042/13059`.
16. R. Lumia, L. Shapiro, O. Zungia. A new connected components algorithms for virtual memory computers. Computer Vision, Graphics and Image Processing (22)2, pp 287-300. 1983.
17. N. Paragios, R. Deriche. Geodesic active regions and level set methods for motion estimation and tracking. Computer Vision and Image Understanding , Volume 97 Issue 3, pp 259-282, 2005.
18. L. Perroton, Segmentation parallèle d'image volumique, ENS Lyon, LIP thesis, France, 1994.
19. W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes in C, The art of scientific computing. second edition, Chapter 1, pp 20-23. Cambridge Press.
20. N. Otsu, A threshold selection method from gray-level histograms. IEEE Transactions on System Man and Cybernetics, 9, pp 62-66.
21. C. Ronse, P.A. Dejvijver. Connected components in binary images: the detection problems. Research Studies Press. 1984.
22. A. Rosenfeld, J.L. Platz. Sequential operator in digital pictures processing, Journal of ACM, vol 13,4, pp 471-494, 1966.
23. S.M. Selkow. One pass complexity analysis of digital pictures properties, Journal of ACM, vol 19,2, pp 283-295, 1972.
24. Y. Shima, T. Murakami, M. Koga, H. Yashiro, H. Fujisawa, A high speed algorithm for propagation-type labeling based on block sorting of runs in binary images. ICPR 1990 pp 655-658.
25. P. Soille, Morphological Image Analysis Principles and applications, p. 38, second edition Springer ISBN 3-540-42988-3, 1999.
26. L. Di Stefano, A simple and efficient connected component labeling algorithm, ICIAP 1999, pp 322-327.
27. K. Suzuki, I. Horiba, N. Sugie, Linear-time connected component labeling based on sequential local operations. Computer Vision and Image Understanding", 89,1 pp 1-23, 2003.
28. K. Wu, E. Otoo, A. Shoshani, Optimizing Connected component labeling algorithms, Pattern Analysis and Applications v11 DOI 10.1007/s10044-008-0109-y, 2008.
29. Y. Yang, D. Zhang, A novel line scan clustering algorithm for identifying connected components in digital images. Image and Vision Computing, DOI: 10.1016/S0662-8856(03)00015-5, 2003.
30. B. Zavidovique, J. Sérot, G. Quénot, Massively parallel dataflow computer dedicated to real time image processing, ICAE 1997, pp 9-29.