Journal of Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Federico Bolelli · Michele Cancilla · Lorenzo Baraldi · Costantino Grana

Towards Reliable Experiments on the Performance of Connected Components Labeling Algorithms

Received: date / Revised: date

Abstract The problem of labeling the connected components of a binary image is well-defined and several proposals have been presented in the past. Since an exact solution to the problem exists, algorithms mainly differ on their execution speed. In this paper, we propose and describe YACCLAB, Yet Another Connected Components Labeling Benchmark. Together with a rich and varied dataset, YACCLAB contains an open source platform to test new proposals and to compare them with publicly available competitors. Textual and graphical outputs are automatically generated for many kinds of tests, which analyze the methods from different perspectives. An extensive set of experiments among state-of-the-art techniques is reported and discussed.

Keywords Connected Components Labeling · Benchmarking, Performance Evaluation

1 Introduction

Part of the responsibility for the huge progress in both computer vision and image processing of the recent years may be credited to the broad access to public image and video datasets. Even if datasets have been blamed for narrowing the focus of research on object recognition, reducing it to a single benchmark performance number, it is now clear that the ability to compare different techniques on the same data allows the reader to choose which algorithm suits his needs best [31]. In Computer Science it is not sufficient to reproduce other people tests, but publishing the source code or, at the very least, an executable should be mandatory. Sometimes, just setting the correct parameters may be a problem, changing the final figures by orders of magnitude. This may be referred to as *reproducible research*, *i.e.* an approach at presenting scientific claims together with all information needed to reproduce the presented results, so that others may verify the findings and build upon them.

Benchmarking may be a problem by itself, because measuring performance may not be obvious, but there are some specific tasks in image processing in which the expected result is known. This reduces the burden of the evaluator, since, after checking that the result is correct, the main question left is to measure how fast an algorithm is.

The problem of labeling the connected components of a binary image is such a problem, so one would expect every paper on the subject to focus on the same evaluation method and data. This is not the case. In recent years, many novel proposals have been published and almost none of them compared on the same data [6, 8, 19].

This paper tackles the problem of evaluating the speed of execution of different strategies to solve the Connected Components Labeling (CCL) problem on binary images.

There are three aspects to keep into account when measuring the "speed of execution" of a family of algorithms: the data on which the algorithms are tested, the hardware capabilities and the quality of the software implementation. Purists may horrify at our omission of computational complexity, but the fact is that CCL is inherently a linear algorithm if we separate it from the equivalence solving (the Union-Find problem), whose computational complexity has been already well studied in depth [30].

The base step for all comparisons is to work on the same data, so our contribution is to provide a public dataset of binary images without any license limitations, or synthetically generated ones. We tried to cover different application scenarios for CCL algorithms such as motion analysis, document processing, OCR, medical imaging, and fingerprint analysis.

Not all computer architectures deliver the same performance on all algorithms and this may also be true for CCL ones. Moreover, the compiler used may significantly impact on the performance of algorithms. The solution

F. Bolelli · M.Cancilla · L. Baraldi · C. Grana
Università degli Studi di Modena e Reggio Emilia
Dipartimento di Ingegneria "Enzo Ferrari"
Tel.: +39-0592056265
Fax: +39-0592056129
E-mail: {name.surname}@unimore.it

we propose to figure out these problems is to provide an open-source *C++* project with a very permissive license, in order to let anyone take the provided algorithms and test them on his own setting, verifying any claim found in the literature (ours included).

Everyone is well aware that software optimization is not an easy task, so the quality of software implementation may change a wonderful algorithm in an unusable junk. Unfortunately, providing source code is not a common requirement for papers to be published, and we had to re-implement many algorithms for which the source code was not available. A positive note is that, being our project open source, any author believing we did him wrong is welcome to provide a better implementation.

The evaluation framework[1] is called Yet Another Connected Components Labeling Benchmark (YACCLAB in short), and the accompanying dataset is the YACCLAB Dataset.

In the following, we will describe the dataset (Section 2), provide some details on how the framework works and how to extend it (Section 3), and summarize the algorithms currently available in YACCLAB (Section 4). An extensive set of experiments is reported in Section 5, discussing and motivating the results. Finally, in Section 6, we draw conclusions.

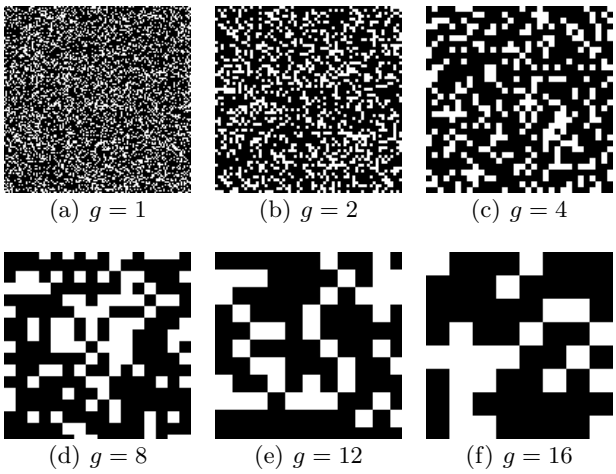
2 The Dataset

Following a common practice in the literature, we built a dataset that includes both synthetic and real images. The provided dataset is suitable for a wide range of applications, ranging from document processing to surveillance, and features a significant variability in terms of resolution, image density, variance of density, and number of components (see Table 1 as a reference). All images are provided in 1 bit per pixel PNG format, with 0 (black) being background and 1 (white) being foreground. The dataset can be automatically downloaded during the set up of the YACCLAB project or it can be found at `http://imagelab.ing.unimore.it/yacclab`.

2.1 Synthetic Images

Random noise images have been used in many papers to evaluate CCL results [13, 17, 18, 22] because connected components in such images have complicated geometrical shapes and complex connectivity to be analyzed. For this reason, we included in our dataset two different set of synthetic images:

1. *Classical:* is the collection of images publicly available in [13], being this the only one already published and used in several other works [9, 19, 29]. These images contain black and white random noise with nine different foreground densities (from 10% up to 90%), and with resolutions ranging from 32×32 to a maximum of 4096×4096 pixels. For every combination of size and density, ten images are provided, making a total of 720 different images. The resulting dataset allows to evaluate performance in terms of scalability on the number of pixels and on the number of labels, which is somehow related to density. For the sake of completeness, this set of images have been generated through the Pseudo Random Number Generator (PRNG) of the *C* standard library implemented in Visual Studio 2008.
2. *Granularity:* encompasses a set of black and white random noise images generated as described by Cabaret *et al.* in [6]. This dataset allows to test algorithms varying not only the pixels density but also their granularity g (*i.e.*, dimension of minimum foreground block), underlying the behaviour of different proposals when the number of provisional labels changes. All the images have a resolution of 2048×2048 and are generated with the Mersenne Twister MT19937 [26] random number generator implemented in the *C++* standard and starting with a *seed* equal to zero. Density of the images ranges from 0% to 100% with step of 1% and for every density value 16 images with pixels blocks of $g \times g$ with $g \in [1, 16]$ are generated. Moreover, the procedure has been repeated 10 times for every couple of density-granularity for a total of 16 160 images.



(a) $g = 1$ (b) $g = 2$ (c) $g = 4$

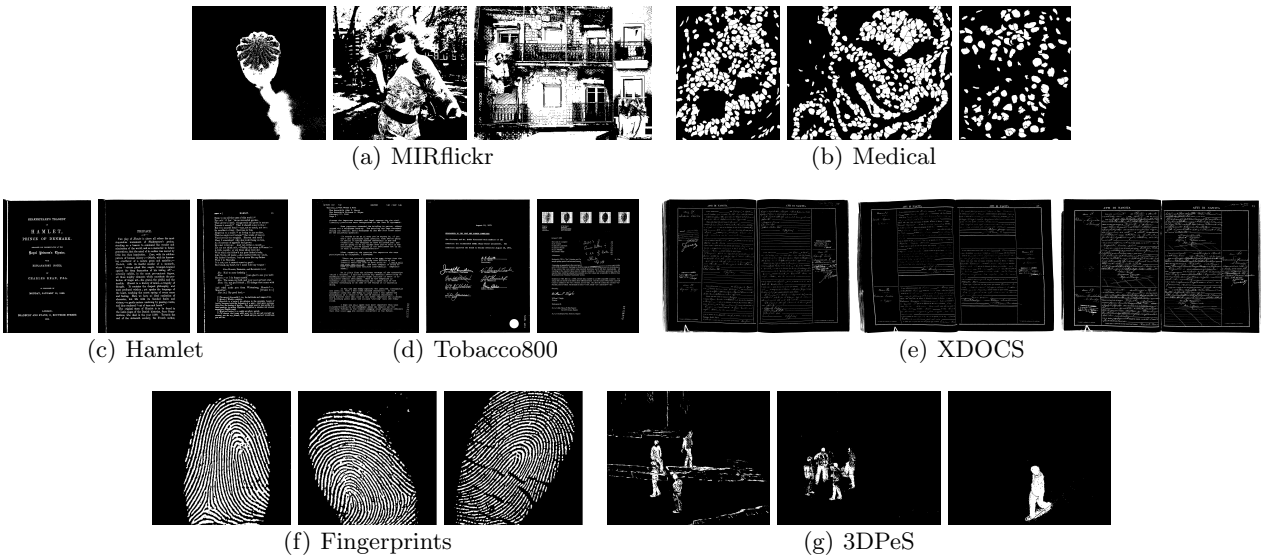
(d) $g = 8$ (e) $g = 12$ (f) $g = 16$

Fig. 1 Sample images taken from the granularity dataset: all images have a foreground density of 30% and a granularity varying from 1 to 16.

2.2 Natural Images

The second set of images we include is the Otsu-binarized [27] version of the MIRflickr dataset [20], publicly avail-

[1] `https://github.com/prittt/YACCLAB`



(a) MIRflickr (b) Medical

(c) Hamlet (d) Tobacco800 (e) XDOCS

(f) Fingerprints (g) 3DPeS

Fig. 2 Sample images from the YACCLAB real datasets.

Table 1 YACCLAB datasets characteristics

	Mpix	*Density*	*Variance*	*#CC*
3DPeS	0.41	0.0263	0.0005	320
Fingerprints	0.14	0.2380	0.0084	809
Hamlet	2.71	0.0789	0.0003	1477
Tobacco800	4.60	0.0475	0.0021	2107
XDOCS	16.49	0.0918	0.0005	15 282
Medical	1.21	0.2469	0.0062	484
MIRflickr	0.18	0.4459	0.0365	492
Classical	2.80	0.4996	0.0669	63 107
Granularity	4.19	0.5000	0.0850	8725

able under a Creative Commons License. It contains 25-000 standard resolution images taken from Flickr, with an average resolution of 0.18 megapixels. There are few connected components (492 on average) and simple patterns, so the labeling is quite easy and fast. Images have an average density of 0.4459 foreground pixels, with a variance of 0.0021. This subset serves again as a comparison with already published results [13, 34].

2.3 Medical Images

Another important task where CCL is an indispensable pre-processing operation is medical image analysis. This dataset, provided by Dong *et al.* [12], is composed of histological images and allows us to cover this fundamental field. Dong *et al.* provide both original images and a binarized version of those images, which are included in the YACCLAB dataset. The process used for nuclei segmentation and binarization is described in [12]. The resulting dataset is a collection of 343 binary histological images with an average amount of 1.21 million pixels to analyze and 484 components to label.

2.4 Document Images

CCL is one of the initial pre-processing steps in most layout analysis or OCR algorithms. Therefore, to cover for Document Analysis applications, three more datasets are added:

1. *Hamlet:* this is a set of 104 images scanned from a version of the Hamlet found on Project Gutenberg[2]. Images have an average amount of 2.71 million of pixels to analyze and 1447 components to label, with an average foreground density of 0.0789.
2. *Tobacco800:* it is composed of 1290 document images and it is a realistic database for document image analysis research, as these documents were collected and scanned using a wide variety of equipment over time. Resolution of documents in Tobacco800 varies significantly from 150 to 300 dpi and the dimensions of images range from 1200 by 1600 up to 2500 by 3200 pixels [1, 23, 24].
3. *XDOCS:* this is a collection of high resolution historical document images taken from the large number of civil registries that are available since the constitution of the Italian state [3, 4, 5]. XDOCS is composed of 1677 images with an average size of 4853×3387 and 15 282 components to analyze. As for most of document dataset, it has a low foreground density of 0.0918.

2.5 Fingerprints Images

This dataset counts 960 fingerprint images collected by using low-cost optical sensors or synthetically generated.

[2] `http://www.gutenberg.org`

Table 2 YAML configuration file of the YACCLAB project.

Parameter name	Description
perform	Dictionary which specifies the kind of tests to perform (correctness, average, average_ws, density and size, granularity and memory).
correctness_tests	Dictionary indicating the kind of correctness tests to perform.
tests_number	Dictionary which sets the number of runs for each test available.
Algorithms	
algorithms	List of algorithms on which apply the chosen tests.
Datasets	
check_datasets	List of datasets on which CCL algorithms should be checked.
average_datasets	List of datasets on which average test should be run.
average_ws_datasets	List of datasets on which average_ws test should be run.
memory_datasets	List of datasets on which memory test should be run.
Utilities	
paths	Dictionary with both input (datasets) and output (results) paths.
write_n_labels	Whether to report the number of connected components in the output files.
color_labels	Whether to output a colored version of labeled images during tests.
save_middle_tests	Dictionary specifying, separately for every test, whether to save the output of single runs, or only a summary of the whole test.

These images were taken from three fingerprint verification competitions (FCV2000, FCV2002 and FCV20040 [25]). In order to fit those images for a CCL application, fingerprints have been binarized using an adaptive threshold [28] and then negated in order to have foreground pixels with value 1. Most of the original images have a resolution of 500 dpi and their dimensions range from 240×320 up to 640×480 pixels.

2.6 Surveillance Images

The final set of images included in YACCLAB comes from a surveillance dataset, namely 3DPeS (3D People Surveillance Dataset [2]). This has been designed mainly for people re-identification in multi-camera systems with non-overlapped fields of view, although it can be exploited for people detection, tracking, action analysis and trajectory analysis. The background models for all cameras are provided by the authors, so a very basic technique of motion segmentation has been applied to generate the foreground binary masks, *i.e.*, background subtraction and Otsu thresholding [27]. The analysis of the foreground masks to remove small connected components and to match the nearest neighbors is a common application for CCL.

3 The Project

YACCLAB is an open source project that enables researchers to test CCL algorithms under extremely variable points of view. The software requires the OpenCV 3.0 library (or higher) to work. A configuration file placed in the installation folder allows the user to specify which kind of tests should be performed, on which datasets, and on which algorithms. A complete description of all configuration parameters is reported in Table 2.

The benchmark provides the following tests, which are deeply described in the following of this Section:

- *correctness*;
- average run-time, also called *average* in the following of the paper;
- average run-time with steps, which will be referred to as *average_ws*;
- *density*, *size* and *granularity* tests on synthetic images;
- memory accesses or simply *memory* test.

It is important to highlight that each test (except correctness and memory ones) is repeated more times per image as specified by the `tests_number` configuration parameter: the minimum execution time for each image is then considered. The use of minimum is justified by the fact that, in theory, an algorithm on a specific environment will always require the same time to execute. This time was computable in exact way on non multitasking single core processors (8086, 80286). Nowadays, too many unpredictable things are happening in the background, independently with respect to the specific algorithm. Anyway, an algorithm cannot use less than the required clock cycles, so the best way to get the "real" execution time is to use the minimum value over multiple runs. The probability of having a higher execution time is then equal for all algorithms. For that reason, taking the minimum is the only way to get reproducible and stable results from one execution of the benchmark to another on the same environment. Two other strategies useful to obtain stable execution times are: i) stopping all the background processes and ii) disabling page swapping during the execution of the benchmark.

YACCLAB has been designed with extensibility in mind, so that new resources can be easily integrated into the project. To introduce a new CCL algorithm into the benchmarking system, it must be compliant with a base interface (Listing 1) implementing the following methods:

```
class Labeling {
public:
    static cv::Mat1b img_;
    cv::Mat1i img_labels_;
    unsigned n_labels_;
    PerformanceEvaluator perf_;

    Labeling() {}
    virtual ~Labeling() = default;

    virtual void PerformLabeling();
    virtual void PerformLabelingWithSteps();
    virtual void PerformLabelingMem(std::vector<unsigned long>& accesses);

    virtual void FreeLabelingData() { img_labels_.release(); }
};
```

Listing 1 Labeling base class from which CCL algorithms have to inherit.

- *PerformLabeling*: includes the whole code of the algorithm and it is necessary to perform average, density, size and granularity tests;
- *PerformLabelingWithSteps*: implements the algorithm, dividing it in steps (*i.e.* `alloc`/`dealloc`, `first_scan` and `second_scan` for those which have two scans, or `all_scan` for the others) in order to evaluate every step separately;
- *PerformLabelingMem*: is an implementation of the algorithm that traces the number of memory accesses whenever they occur.

The *C++* savvy will notice the fact that the labeling methods are declared virtual, thus adding an overhead to the function call. We measured the impact of this and verified that it is many orders of magnitude lower than the time required by the algorithms, being in fact negligible. Additionally, it is the same for all algorithms.

All CCL algorithms included in YACCLAB implement the 8-way connectivity. Moreover, in order to compare different proposals as fairly as possible, we standardized shared code preferring, for example, the *new* statement to allocate memory or using, when possible, the same data types.

We look at YACCLAB as a growing effort towards better reproducibility of CCL algorithms, so implementations of new and existing labeling methods are welcome.

3.1 Correctness Test

The first kind of test that YACCLAB enables is related to correctness. In order to check the correctness of an implementation, indeed, the output of an algorithm is compared with that of the Scan plus Array-based Union-Find algorithm [33], which is assumed to be a correct reference point. Before making the comparison, labels indexes are changed to force a row major ordering: different labeling paradigms may assign different labels to the same object and this is not considered an error. The datasets on which correctness test shall be executed have to be specified through the `check_datasets` list in the configuration file. An additional dataset to the ones already described in Section 2 has been specifically designed for correctness tests. This collection, called *check* and included in YACCLAB, contains a set of 20 binary images with special sizes (*i.e.* odd number of rows/columns or single row/column) and a chessboard texture to test algorithms in the most critical cases.

3.2 Average Run-Time Test

Average run-time test executes algorithms on every image of a specified dataset and reports the average execution times in three different formats: plain text files, histogram charts, either in color or in gray-scale, and LaTeX tables, which can be directly included in research papers.

Such kind of test can be applied on all YACCLAB datasets and the `average_datasets` list enables the user to specify which ones should be selected. YACCLAB performs average test only whether the associated entry in the `perform` dictionary of the configuration file is set to "true".

It should be noted that the execution times calculated by average test always include memory allocation. We strongly believe that excluding memory allocation from the execution time is not impartial. Indeed, each algorithm may require a different number of tables and obviously these impact on performance. In real applications, CCL is applied on images of different sizes and this requires to reallocate data when needed. For this reason, it is mandatory to include the memory allocation time to

evaluate the performance of an algorithm. On the other hand, there are cases in which it could be fair to compare algorithms without considering memory allocation: in an embedded system in which images are always captured with the same size, for example, could be realistic to allocate memory only once. In order to cover these circumstances, we introduced the average_ws test in our benchmarking system, as described in Section 3.3.

3.3 Average Run-Time Test with Steps

This test evaluates the performance of an algorithm separating the allocation/deallocation time (*alloc*/*dealloc*) from the time required to compute labeling. Moreover, if an algorithm employs multiple scans to produce the correct output labels, YACCLAB will store the time of every scan and will display them separately.

To understand how YACCLAB computes the memory allocation time for an algorithm on a reference image, it is important to underline the subtleties involved in the allocation process. Indeed, all modern operating systems (not real-time, nor embedded ones, but certainly Windows and Unix) handle virtual memory exploiting a *demand paging* technique, *i.e.* demand paging with no pre-paging for most of Unix OS and cluster demand paging for Windows OS. This means that a disk page is copied into physical memory only when it is accessed by a process the first time, and not when the allocation function is called. Therefore, it is not possible to calculate the exact allocation time required by an algorithm, which computes CCL on a reference image, but its upper bound can be estimated using the following approach:

1. forcing the allocation of the entire memory by reserving it (*malloc*), filling it with zeros (*memset*), and tracing the time;
2. calculating the time required by the assignment operation (*memset*), and subtracting it from the one obtained at 1;
3. repeating the previous points for all data structures needed by an algorithm and summing times together.

This will produce an upper bound of the allocation time because caches may reduce the second assignment operation, increasing the estimated allocation time. Moreover, in real cases, CCL algorithms may reserve more memory than they really need, but the *demand paging*, differently from our measuring system, will allocate only the accessed pages.

3.4 Synthetic Test

Test on synthetic images are useful to evaluate the performance of different approaches in term of scalability on the number of pixels and labels. Moreover, synthetic tests give us the possibility to highlight the behavior of CCL algorithms when the foreground pixels density changes. YACCLAB divides this test into two groups:

- density and size: are performed on *classical* dataset (Section 2.1) and estimate the performance of different CCL algorithms when they are executed on images with varying foreground density and size. The density test calculates the mean execution time of each algorithm over images whose density ranges from 10% up to 90%, with a 10% step. On the other hand, size test reports average execution time on images having resolutions ranging from 32×32 up to 4096×4096;
- granularity: evaluates an algorithm varying density (from 1% to 100%, using a 1% step) and pixels granularity, but not images resolution. This test was proposed in [6] and its inclusion in YACCLAB allows to also verify the performance claims of CCL algorithms on this demanding case. The output results display the average execution time over images with the same density and granularity.

3.5 Memory Accesses Test

This test is useful to correlate the performance of an algorithm to the number of memory accesses. The memory test computes the average number of accesses to the label image (*i.e.* the images usually used to store the provisional and then the final labels for the connected components), the average number of accesses to the binary image to be labeled and, finally, the average number of accesses to data structures used to solve the equivalences between label classes. Moreover, if an algorithm requires extra data, memory test summarize them as "other" accesses and returns the average. Furthermore, all contributions of each algorithm and dataset are summed together in order to show the total amount.

Since counting the number of memory accesses imposes additional computations, the code implementing this test is not shared with that implementing the others.

3.6 NULL labeling

The *NULL labeling* is a new "algorithm" which defines a lower bound limit for the execution time of CCL on

Algorithm 1: NULL labeling

Input: IN, binary image of width w and height h
Result: OUT, integer image of width w and height h

```
for r ← 0 to h − 1 do
    for c ← 0 to w − 1 do
        OUT(r, c) = IN(r, c)
    end
end
```

a given machine and dataset. As the name suggests, the *NULL labeling* does not provide the correct connected components for a given image. It only copies the pixels from the input image to the output one.

It is important to underline that the lower bound limit defined by the *NULL labeling* can not be overtaken by any CCL algorithm. The operations performed by NULL labeling allow to identify the minimum time required for allocating the memory of the output image, reading the input image and writing the output one. In this way, all the algorithms may be compared in terms of how costly are the additional operations required.

3.7 Union-Find Templating

Following the idea of extensibility, an algorithm can also be templated on the Union-Find strategy. YACCLAB is able to compare each algorithm (but those for which the labels solver is built-in) with four different labels solving strategies: standard Union-Find (UF), Union-Find with Path Compression (UFPC) [32], Interleaved Rems algorithm with splicing (RemSP) [11] and Three Table Array (TTA) proposed in [16]. This flexibility allows YACCLAB to better analyze the behavior of a specific algorithm and the user to choose the best combination for his machine, compiler, and operating system. The standardization of the Union-Find algorithms reduces the code variability and provides fairer comparisons. Particular care was used to check that the flexibility introduced with Union-Find templating did not impact on execution time. We compared the produced code at assembly level and the output is the same with and without templates.

4 Available Algorithms

Since version 3.0, OpenCV included CCL features. The algorithm implemented was the one described in [32, 33], which is basically equivalent to the one in [18][3]. It uses a pixel based scanning with online equivalence solving by means of a Union-Find technique with path compression, plus a decision tree for accessing only the minimum number of already labeled pixels.

Still, this is the reference algorithm implemented in YACCLAB, because of its very good performance and ease of understanding.

In [13] we proved that different versions of the decision tree are equivalent to the previous one and extended it to Block Based scanning, that is scanning the image in 2×2 blocks. Building the decision tree for that case is much harder, because of the large number of possible combinations. In [14] we proposed a proved optimal strategy to build the decision tree by means of a dynamic programming approach. In YACCLAB we provide an implementation of this algorithm which employs the usual OpenCV optimizations on image accesses.

Another variation of Block Based analysis was proposed in [9], which is reported to be faster than the previous one. We also include this algorithm thanks to the availability of the source code on the web pages of authors, making the signature compliant with our standards and adding some checks to deal with images with an odd number of rows and columns as the other algorithms do.

In [21] a different take on labeling, called Light Speed Labeling, was proposed. The paper has a well described pseudocode for the algorithm, and in the journal version [6] further analyses have been performed, also proposing some variations and stating that it is the fastest algorithm available. To our knowledge, no public implementation exists, so we are the first ones to really make it available. This is probably due to two small mistakes in the pseudocode of [22]: a w was called n because of a change in notation between the two papers and a "step 2" was missing in a "for" loop. We implemented both the standard version —this is the name used by the authors—, the compressed version, and the *zero-offset* optimization, that is reported to provide a speedup of up to 5%.

In order to demonstrate the importance of the algorithm used to solve label equivalences, we include an implementation of [10] which uses a two scan procedure with an online labels solver algorithm (exploiting an array-based structure to store the label equivalences). This technique requires multiple searches over the array at every Union operation, leading to a clear non-linear behavior with respect to the number of labels.

As a representative of the contour tracing type of algorithms we include the approach proposed in [7]. This approach clockwise tags all pixels in both the contour and the immediately external background area in a single operation. Then, during the raster scan, when an untagged boundary is found, a counterclockwise contour tracing is performed for internal contours. This technique proved to have a linear complexity with respect to the number of labels and run quite fast, also because the filling of the connected components (label propagation after contour following) is cache-friendly for images stored in a raster scan order.

In order to cover a new recent paradigm for CCL, YACCLAB includes two more algorithms: Configuration-Transition-Based [19] originally proposed by He *et al.* in 2014 and Optimized Pixel Prediction [15] proposed by Grana *et al.* in 2016.

The algorithm from He —using a reduced version of the 2×2 scan mask, from the Block Based algorithm, to just 1×2 pixels— scans the given image at alternate lines and processes pixels two by two. The authors also define nine different configurations and nine groups of as-

[3] After the appearance of the YACCLAB project results, OpenCV changed its default algorithm to the fastest one reported in YACCLAB.

sociated actions for the pixels in the current scan mask. Then, in the labeling approach, the next-case decisions are represented as a configuration-transition diagram to obtain better speed. The current *state* of the algorithm is defined by the values of pixels already checked in the current scan mask, and by previous actions executed. This approach allows He *et al.* to save the number of accesses to pixels and to speed up the labeling process. The design of the algorithm is specific for the CCL problem and there is no prevision of extending it to any other similar task.

In Optimized Pixel Prediction, instead, we proposed a general paradigm to exploit already seen pixels during the scan phase, in order to minimize the number of times a pixel is accessed. As shown in literature, the decision table which rules the scan step can be conveniently converted to an optimal binary decision tree [13], in which internal nodes represent conditions on mask pixels, and leaves represent actions to be performed on the current pixel of the provisional labels image. Usually, the same decision tree is traversed for each pixel of the input image, without exploiting values seen in the previous iteration, which, if considered, would result in a simplification of the decision tree for that pixel. To go beyond this limitation, we computed a reduced decision tree for each possible set of known pixels: these reduced decision trees are then connected into a single graph, which rules the execution of the CCL algorithm on the whole image. Each leaf of a tree, which represents the action to be performed on a pixel, is connected to the root of a second tree, which should be executed for the next pixel. The obtained graph can then be directly converted into running code.

Finally, we also include the stripe-based algorithm proposed by Zhao *et al.* [34]. Here, each pair of consecutive rows in the image is taken as a work-region (called *stripe*). Foreground pixels in each stripe are replaced with a global position value (the index of the leftmost pixel of the stripe connected to that pixel). Every stripe is then considered as a rooted tree, and the image is abstracted as the forest composed of all stripes. Stripes are therefore merged together by merging multiple rooted trees. Finally, the image is traversed to find the roots of non-zero pixels and obtain label values, which are then assigned to the non-zero pixels. In the original paper, the number of rows in the image is supposed to be even, and authors state that no extra auxiliary memory is required. We notice that to store global position values on images having more than 256 labels, pixels should be converted to a larger data type, thus requiring additional memory. To make a fair comparison, our implementation copies the input image to one with 32 bits per pixel, and adds proper checks to deal with the last row.

5 Experimental Results

In this Section, experimental results obtained with YACCLAB over the aforementioned algorithms are presented and analyzed.

There are many variables that may significantly influence the performance of an algorithm: the chosen labels solver, the adopted compiler, the operating system on which tests are performed, the machine architecture and last but not least the data on which algorithms are executed. Unfortunately, all these combinations generate too much data to be analyzed and reported in a single paper, so that we select the most significant and general ones, highlighting the strengths and weaknesses of the state-of-the-art algorithms.

Specifically, we picked nine different algorithms: five of them can use four different labels solving algorithms, one (Light Speed Labeling) has four variations and each can use two labels solving algorithms, for a total of 31 combinations plus the NULL labeling. We have seven datasets for the average run-time test, for a total of 224 values, which we fit in a single results table. We considered two compilers (MSVC 19.11.25508.2 with 02 speed optimizations enabled and GCC 5.1.0 with 03 optimization flag) and two operating systems (Windows 10.0.15063 64 bit and Linux 4.10.0-33-generic 64 bit), for a total of three possible combinations (the Microsoft compiler cannot run under Linux). We selected a single machine in order to showcase the performance of the different algorithms: an Intel Core i7-4770 CPU @ 3,40 GHz (with 4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache), and with 24 GB of RAM available. We understand that a comparison in performance between different machines would be useful, but it would require to replicate all the considerations, leaving anyway space to the question whether the observations on the two selected machines could be applied also to another one.

In the following, we will use acronyms to refer to the available algorithms: CT is the Contour Tracing approach by Fu Chang *et al.* [7], CCIT is the algorithm by Wan-Yu Chang *et al.* [9], DiStefano is the algorithm in [10], BBDT[4] is the Block Based with Decision Trees algorithm by Grana *et al.* [13], SAUF[4] is the Scan plus Array-based Union-Find algorithm by Wu *et al.* [33], CTB is the Configuration-Transition-Based algorithm by He *et al.* [19], SBLA is the stripe-based algorithm by Zhao *et al.* [34], and PRED is the Optimized Pixel Prediction by Grana *et al.* [15]. Additionally, the four different versions of the Light Speed Labeling algorithm by Lacassagne *et al.* [6] are identified as LSL_STD, LSL_STDZ, LSL_RLE, and LSL_RLEZ, where STD refers to the standard version of the algorithm, RLE refers to the run length encoding optimization and Z is related to the *zero-offset* addressing optimization as described by the au-

[4] SAUF and BBDT are the algorithms currently included in the OpenCV library (since version 3.2).

Table 3 Average run-time results in ms obtained under Windows 10 with MSVC 19.11.25508.2 compiler. The bold values represent the best labels solver for a specific CCL algorithm and dataset, the red ones point out the best algorithm for a given dataset.

	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
SAUF_RemSP	0.823	**0.401**	6.206	2.817	**0.506**	9.648	**48.605**
SAUF_TTA	0.830	0.401	6.245	2.849	0.507	9.698	48.959
SAUF_UFPC	**0.823**	0.402	6.212	2.824	0.510	9.657	48.673
SAUF_UF	0.824	0.401	**6.205**	**2.814**	0.510	**9.647**	48.638
BBDT_RemSP	0.650	**0.314**	**5.011**	**2.186**	**0.396**	**7.951**	**40.587**
BBDT_TTA	0.660	0.316	5.038	2.187	0.397	7.995	40.789
BBDT_UFPC	0.651	0.315	5.023	2.203	0.397	7.961	40.684
BBDT_UF	**0.650**	0.317	5.033	2.199	0.397	7.965	40.658
CCIT_RemSP	0.779	0.356	5.898	2.600	0.461	9.389	46.675
CCIT_TTA	**0.741**	**0.348**	**5.666**	**2.520**	**0.451**	**8.923**	**45.271**
CCIT_UFPC	0.741	0.370	5.813	2.634	0.486	9.025	46.227
CCIT_UF	0.763	0.363	5.876	2.598	0.471	9.238	46.633
CTB_RemSP	**0.809**	**0.377**	**6.100**	**2.810**	**0.493**	**9.659**	**48.375**
CTB_TTA	0.941	0.407	7.240	3.184	0.524	11.741	55.597
CTB_UFPC	0.811	0.381	6.111	2.804	0.498	9.643	48.446
CTB_UF	0.892	0.404	7.298	3.147	0.534	11.679	56.029
PRED_RemSP	**0.720**	**0.349**	**5.475**	**2.534**	**0.467**	**8.499**	**44.642**
PRED_TTA	0.730	0.353	5.539	2.603	0.477	8.578	45.736
PRED_UFPC	0.723	0.355	5.496	2.579	0.479	8.528	44.854
PRED_UF	0.723	0.353	5.492	2.573	0.480	8.532	45.408
LSL_STD_TTA	2.149	0.697	16.503	7.136	0.994	27.843	127.648
LSL_STD_UF	**2.139**	**0.694**	**16.466**	**7.122**	**0.991**	**27.793**	**127.433**
LSL_STDZ_TTA	1.716	0.551	13.459	5.802	0.802	22.696	104.932
LSL_STDZ_UF	**1.707**	**0.550**	**13.411**	**5.784**	**0.800**	**22.650**	**104.850**
LSL_RLE_TTA	1.695	0.631	13.722	5.880	0.846	22.600	105.326
LSL_RLE_UF	**1.682**	**0.629**	**13.637**	**5.843**	**0.843**	**22.517**	**104.901**
LSL_RLEZ_TTA	1.767	**0.654**	**14.216**	**6.092**	**0.876**	23.441	**108.819**
LSL_RLEZ_UF	**1.760**	0.654	14.223	6.108	0.878	**23.428**	108.944
DiStefano	0.881	0.606	7.556	4.037	0.850	11.363	90.903
CT	0.988	0.937	9.684	4.517	0.938	12.820	75.670
SBLA	0.878	0.551	7.480	3.559	0.666	10.887	61.673
NULL	0.369	0.097	2.508	1.120	0.165	4.414	21.579

thors in the paper. Moreover, NULL is the lower bound limit described in Section 3.6. Finally, to identify the labels solver adopted to test an algorithm, the acronyms already presented in Section 3.7 are placed after its name.

5.1 Average Run-Time Test Results

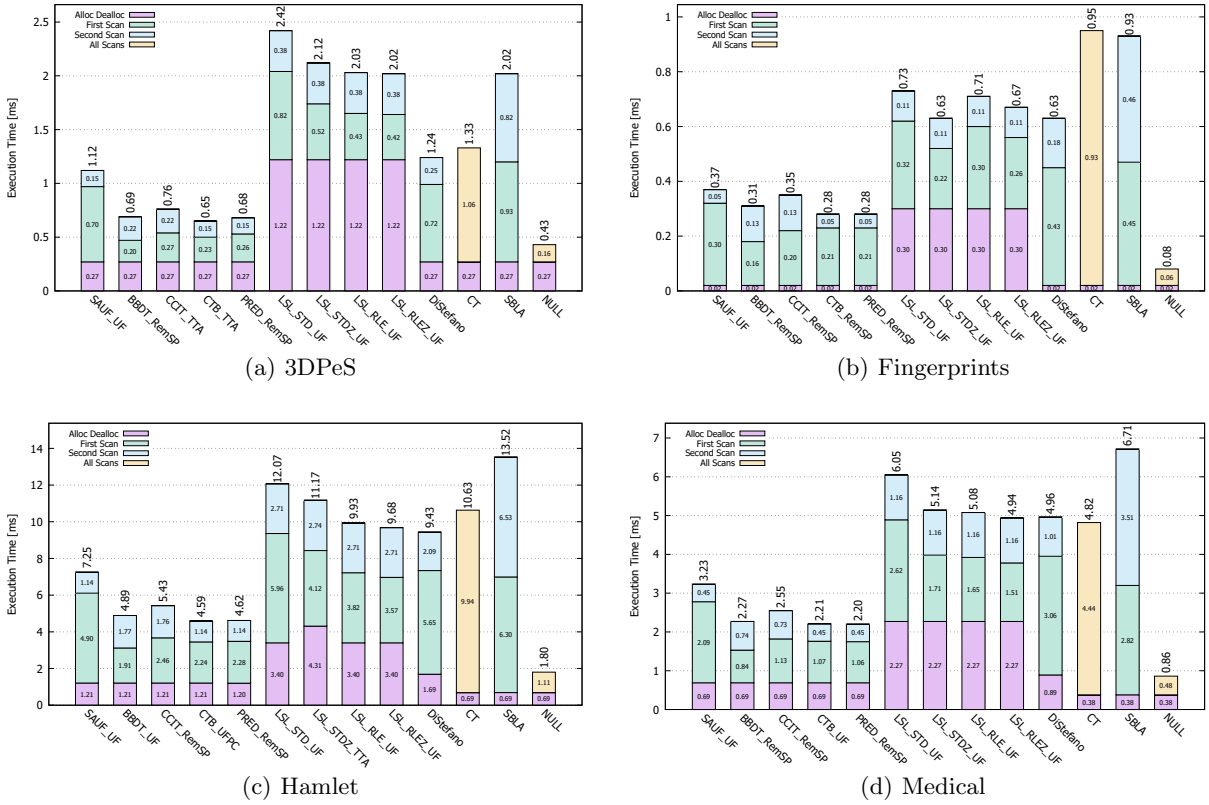
Results of average tests are summarized in Tables 3, 4(a), and 4(b) (the last two are at the end of the paper).

The first conclusion we can draw is that Linux is faster for this task. Extremely so. In particular, when memory allocation becomes important, *i.e.* when the dataset has large images, or the algorithm requires larger data structures, the time required by Windows may be double than Linux. Other tests reported the cause to be a higher allocation time, due to the virtual page commits.

In order to evaluate how labels solver algorithms affect the performance, we estimated the time required by all of them for every algorithm and dataset. Changing the labels solver can lead to significant enhancements for specific combinations of algorithm, data, and operating system, or it can make no difference for others. It appears that the best strategy the user can follow is to test the algorithms on his specific configuration and pick the one which delivers the best performance.

For what concerns labels solver algorithms, the following tentative conclusions can be drawn:

- Windows MSCV: RemSP provides the best performance with all algorithms, except with CCIT that uses TTA for best results.
- Windows GCC: UF provides the best performance with all algorithms, or it is equivalent to RemSP with CCIT.
- Linux GCC: TTA and RemSP are almost equivalent, and the others are not too different, except, again, with CCIT that significantly favors TTA.



(a) 3DPeS (b) Fingerprints

(c) Hamlet (d) Medical

Fig. 3 Average run-time tests with steps in ms on an Intel Core i7-4770 *CPU* @ 3.40GHz running Linux with GCC 5.1.0 (lower is better).

The different memory management, between the two O.S., can again be a possible explanation of performance variance between labels solvers. Linux seems to have a better performance, since it allocates memory pages with a speed that doubles the Windows memory allocation capability. TTA is, at least in theory, really efficient when merges are encountered but, unfortunately, this efficiency requires more memory. Therefore, the advantages appear significant under Linux, instead, with Windows, they are frustrated by the allocation costs.

For all we said, expressing a judgment on which algorithm is "best" is extremely difficult, and maybe plain wrong. Under Windows, BBDT has best performance, irrespective of the compiler. Under Linux, CTB demonstrates, in most cases, the best behavior. This is true for our test machine, and can be justified later by observing the behavior of other tests, but we cannot say which algorithm of the two is the fastest.

For what regards LSL, our tests do not confirm the results on CCL presented in [6], but it is clear that the *zero-offset* optimization is extremely beneficial for LSL_STD, less so for LSL_RLE. The RLE version is most of the times faster than the STD one, even after the *zero-offset* optimization.

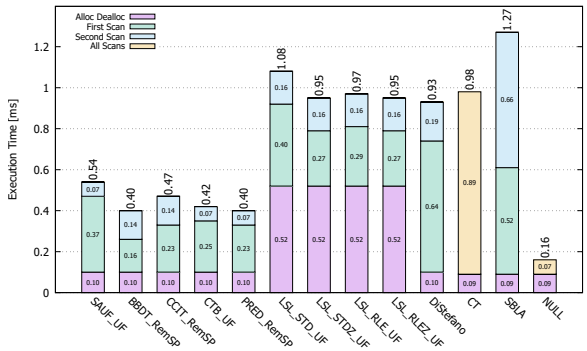
5.2 Average Run-Time Test with Steps Results

In order to discuss the single phases of each algorithm, we need to shorten the algorithm lists. We thus select, for each algorithm and dataset combination, the labels solver which has the lowest total execution time when using the *PerformLabelingWithSteps* methods. This also allows to show the charts produced by YACCLAB in Fig. 3 and Fig. 4.

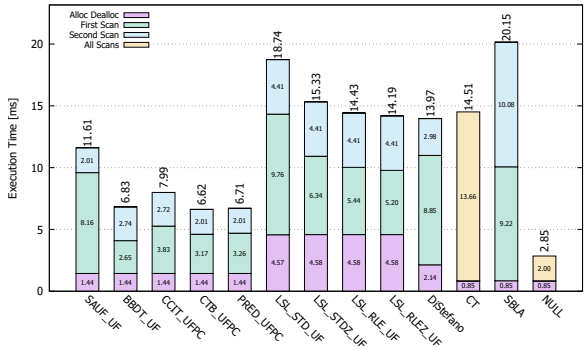
The allocation/deallocation time is stable and depends only on the data structures used:

- CT, SBLA, and NULL do not have any memory requirement in addition to the output image;
- SAUF, BBDT, CCIT, CTB, and PRED have the additional requirements of the Union-Find structures, which are one (UF, UFPC, RemSP) or three arrays (TTA);
- DiStefano has a two arrays structure to handle labels resolution;
- LSL has always a larger memory footprint.

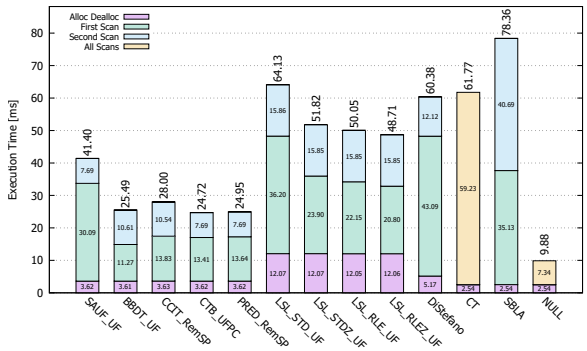
All this is reflected in the time requirements. Note that the time is quantized on the number of virtual memory pages required, so for example on 3DPeS or Fingerprints, there is no difference but for LSL.



(a) MIRflickr



(b) Tobacco800



(c) XDOCS

Fig. 4 Average run-time tests with steps in ms on an Intel Core i7-4770 *CPU* @ 3.40GHz running Linux with GCC 5.1.0 (lower is better).

The first scan of BBDT is always faster than that of the other two scan algorithms, while its second scan is slower. This is clearly due to the fact that the 2×2 mask used requires a bunch of tests in the second scan, which are saved in the first scan. CCIT has exactly the same second scan and the same timings, but its first scan is slower due to a different organization of the decision tree. LSL second scan has to make two indirections and this causes another slowdown. Of course the real problem of LSL is the first scan, which is extremely costly and slower than the other efficient algorithms.

The memory savings of SBLA are annihilated by the horrible cache accesses caused by the simultaneous use of the output image as a Union-Find structure. CT is heavily affected by the length of the contours, so its worse performance is obtained on the fingerprints dataset.

5.3 Memory Test Results

YACCLAB can also report the number of memory accesses of each algorithm, for each dataset. We are not reporting these numbers for all combinations, but we simply try to graphically report a summary in Fig. 5, which has been computed under Linux, with GCC, on Tobacco800 dataset, using the classical UF algorithm. The chart compares memory accesses and execution time, but normalizes them with reference to the values of NULL labeling. For this reason, the axes start at 1 both for x and y and the values are adimensional.

There is a correlation between number of memory accesses and time, but it is not linear and not perfect. In fact, CT has very few memory accesses, similarly to CCIT, but their access patterns differ a lot: CT is definitely not cache friendly and thus CCIT is much faster. Furthermore, BBDT uses the same mask of CCIT, requires slightly more accesses, but it is faster than CCIT because the same results are obtained with a more structured and regular code which is easier for the compiler to optimize. CTB and PRED instead use a smaller mask, thus requiring more accesses to both the output image, and the equivalence storage structures. Their mask is anyway used smartly, containing the number of accesses to the input image (they leverage the outcome of one

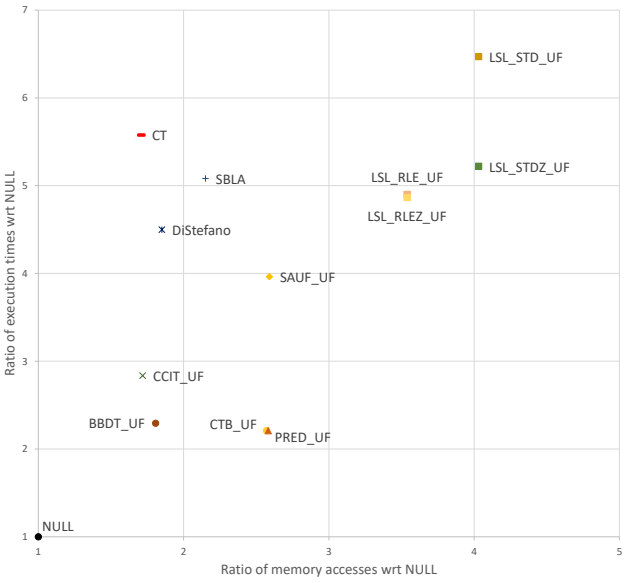
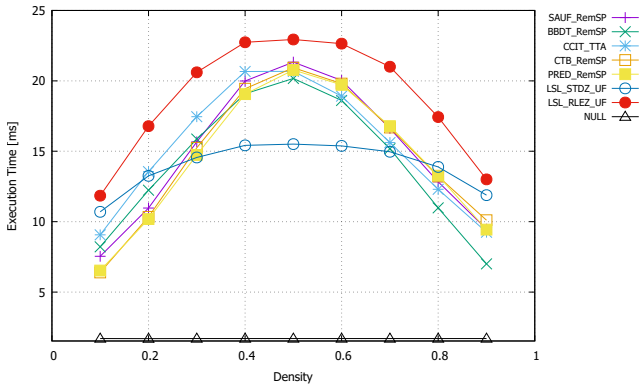
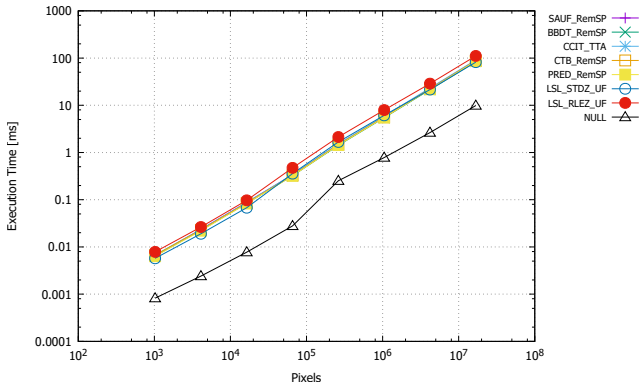


Fig. 5 Correlation between memory accesses and execution times normalized to the NULL labeling and obtained under Linux with GCC 5.1.0 on Tobacco800 dataset.



(a) Density



(b) Size

Fig. 6 Size and Density tests on Intel Core i7-4770 *CPU* @ 3.40GHz running Linux with GCC 5.1.0. Some algorithms have been omitted to make the charts more legible.

pixel to evaluate the next one), thus allowing for an extremely simple and branchless second scan. As already analyzed before, DiStefano and SBLA have extremely poor labels solving strategy which causes the execution time to be very high. The other algorithms have even higher memory requirements, which cause more slowdowns and impact on performance.

5.4 Synthetic Tests Results

The final results have been obtained on the synthetic datasets and are reported in Fig. 6 and Fig. 7. Again, these tests are reported for a reduced set of combinations, selecting the best performing labels solver, under Linux and with GCC. We removed the worse performing algorithms in order to keep the figure readable. The *Density* chart shows how, depending on the number of foreground pixels, the behavior changes. This is mostly related to the errors of the branch prediction unit which heavily affects the algorithms around 0.5. It is interesting to note that PRED and CTB behave extremely similarly and are effective at low densities, LSL_STD is much less affected by the density changes, BBDT is faster at higher densities.

We included these tests in YACCLAB for the sake of completeness, but during the writing of this paper we realized *why* virtually any conclusion drawn from this is inapplicable to real world images. The problem is that the density of foreground pixels is not the key factor in the branch prediction: the key point is the probability of transition from black to white and viceversa. We are keeping it fixed at 50% using random generation, but in real images this is definitely not the case. So the branch predictor has much less problems and the result are more similar to those observed at lower densities.

Less interesting is the *Size* chart, which shows that the behavior is linear in the number of pixels for all the selected algorithms, as expected. The only point we want to stress is the "jump" observed around 10^5 pixels. This is due to the images not fitting L2 cache anymore and requiring also L3 cache to be employed to fit the input image.

The *Granularity* charts, on the other hand, highlight the performance of an algorithm when both density of foreground pixels and their granularity change. Of course, when the granularity is equal to 1 (Fig. 7(a)) the algorithms performance have the pattern already observed in the density chart. Then, when the granularity grows, the execution time for middle density images decrease. Again, this can be easily explained considering the branch prediction unit: when pixel blocks in the input image became bigger, the prediction of pixel values, which are totally random, fails less, thus decreasing the cost associated to failures.

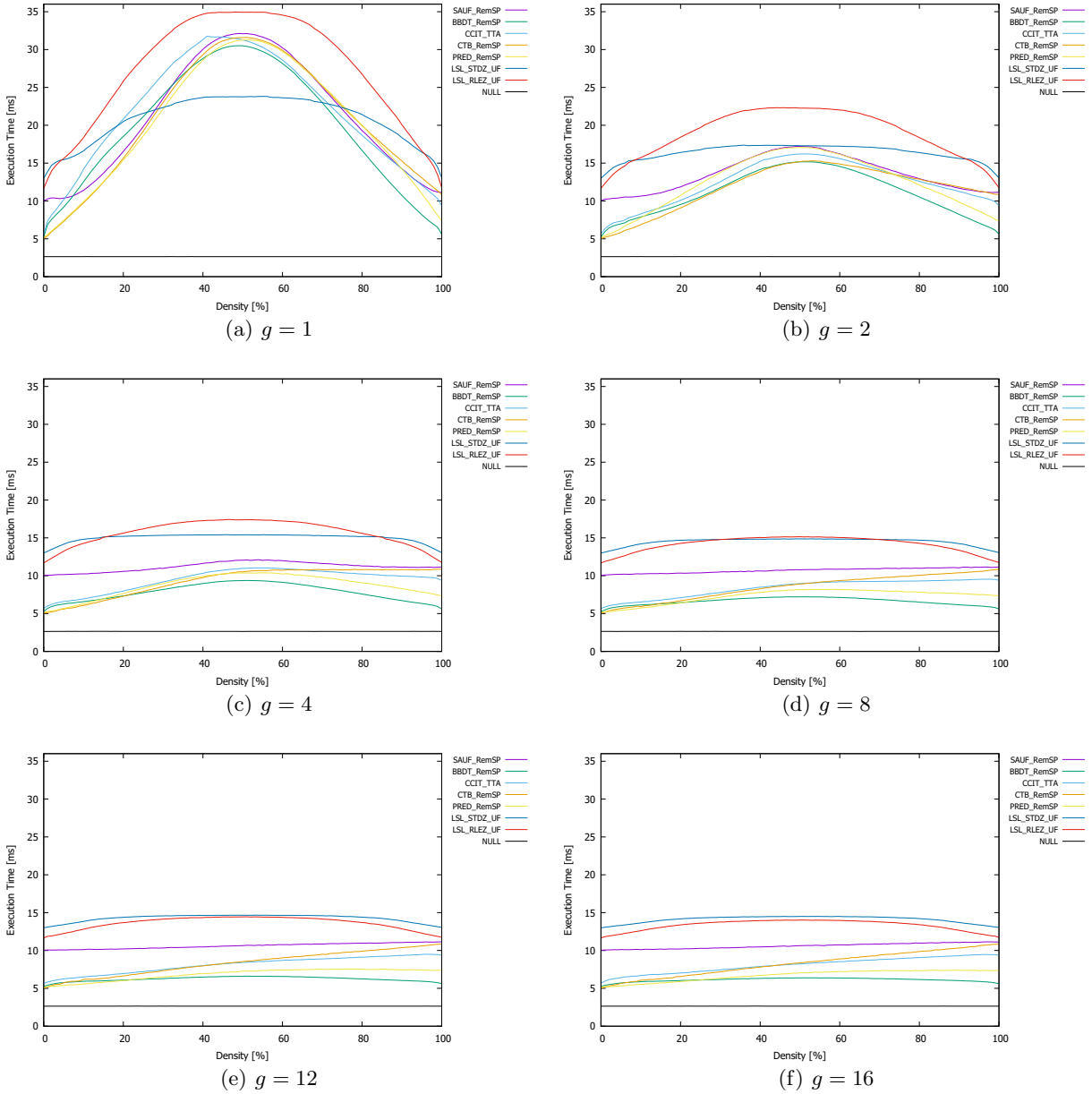
6 Conclusions

In this paper we described two contributions to the image processing community: a comprehensive dataset for comparing connected components labeling algorithms and a portable open source *C++* project to test different algorithms on top of it. No new algorithms were proposed, but this tool allows any new improvement to be evaluated uniformly with respect to existing proposals.

We presented an analysis of some results to showcase the possibility of this project, and doing so we demonstrated that in many cases it is impossible to find an algorithm which clearly dominates the others. Moreover, giving normalized figures such as *clock per pixel* is poorly significant, because it implies that that number does not depend on the machine. When changing the compiler changes the algorithm behavior, the number becomes clearly useless.

This is not to say that no comparison is possible. If an algorithm is always faster than another in all tried configurations, the conclusion is then obvious.

The *reproducible research* movement is strongly affecting the Image Processing community, making sci-



(a) $g = 1$ (b) $g = 2$

(c) $g = 4$ (d) $g = 8$

(e) $g = 12$ (f) $g = 16$

Fig. 7 Granularity results in ms on an Intel Core i7-4770 *CPU* @ 3.40GHz running Linux with GCC 5.1.0 (lower is better).

entific advances readily available to all researchers and practitioners. We strongly support this view and our effort is exactly aiming at letting everybody pick the best CCL algorithm for his needs.

References

1. Agam G, Argamon S, Frieder O, Grossman D, Lewis D (2006) The Complex Document Image Processing (CDIP) Test Collection Project. Illinois Institute of Technology
2. Baltieri D, Vezzani R, Cucchiara R (2011) 3DPeS: 3D People Dataset for Surveillance and Forensics. In: Proceedings of the 2011 joint ACM workshop on Human gesture and behavior understanding, ACM, pp 59–64
3. Bolelli F (2017) Indexing of historical document images: Ad hoc dewarping technique for handwritten text. In: 13th Italian Research Conference on Digital Libraries
4. Bolelli F, Borghi G, Grana C (2017) Historical handwritten text images word spotting through sliding window hog features. In: 19th International Conference on Image Analysis and Processing

5. Bolelli F, Borghi G, Grana C (2018) Xdocs: An application to index historical documents. In: Italian Research Conference on Digital Libraries, Springer, pp 151–162
6. Cabaret L, Lacassagne L, Etiemble D (2016) Parallel light speed labeling: an efficient connected component algorithm for labeling and analysis on multi-core processors. Journal of Real-Time Image Processing pp 1–24
7. Chang F, Chen CJ, Lu CJ (2004) A linear-time component-labeling algorithm using contour tracing technique. Computer Vision and Image Understanding 93(2):206–220
8. Chang WY, Chiu CC (2014) An efficient scan algorithm for block-based connected component labeling. In: 22nd Mediterranean Conference of Control and Automation (MED), IEEE, pp 1008–1013
9. Chang WY, Chiu CC, Yang JH (2015) Block-based connected-component labeling algorithm using binary decision trees. Sensors 15(9):23,763–23,787
10. Di Stefano L, Bulgarelli A (1999) A Simple and Efficient Connected Components Labeling Algorithm. In: International Conference on Image Analysis and Processing, IEEE, pp 322–327
11. Dijkstra EW (1976) A discipline of programming / Edsger W. Dijkstra. Prentice-Hall Englewood Cliffs, N.J
12. Dong F, Irshad H, Oh EY, Lerwill MF, Brachtel EF, Jones NC, Knoblauch NW, Montaser-Kouhsari L, Johnson NB, Rao LK, et al (2014) Computational Pathology to Discriminate Benign from Malignant Intraductal Proliferations of the Breast. PloS one 9(12):e114,885
13. Grana C, Borghesani D, Cucchiara R (2010) Optimized Block-based Connected Components Labeling with Decision Trees. IEEE Transactions on Image Processing 19(6):1596–1609
14. Grana C, Montangero M, Borghesani D (2012) Optimal decision trees for local image processing algorithms. Pattern Recognition Letters 33(16):2302–2310
15. Grana C, Baraldi L, Bolelli F (2016) Optimized Connected Components Labeling with Pixel Prediction. In: Advanced Concepts for Intelligent Vision Systems
16. He L, Chao Y, Suzuki K (2007) A Linear-Time Two-Scan Labeling Algorithm. In: International Conference on Image Processing, vol 5, pp 241–244
17. He L, Chao Y, Suzuki K (2008) A Run-Based Two-Scan Labeling Algorithm. IEEE Transactions on Image Processing 17(5):749–756
18. He L, Chao Y, Suzuki K, Wu K (2009) Fast connected-component labeling. Pattern Recognition 42(9):1977–1987
19. He L, Zhao X, Chao Y, Suzuki K (2014) Configuration-Transition-Based Connected-Component Labeling. IEEE Transactions on Image Processing 23(2):943–951
20. Huiskes MJ, Lew MS (2008) The MIR Flickr Retrieval Evaluation. In: MIR '08: Proceedings of the 2008 ACM International Conference on Multimedia Information Retrieval, ACM, New York, NY, USA
21. Lacassagne L, Zavidovique B (2009) Light Speed Labeling for RISC architectures. In: ICIP, pp 3245–3248
22. Lacassagne L, Zavidovique B (2011) Light speed labeling: efficient connected component labeling on risc architectures. Journal of Real-Time Image Processing 6(2):117–135
23. Lewis D, Agam G, Argamon S, Frieder O, Grossman D, Heard J (2006) Building a test collection for complex document information processing. In: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, pp 665–666
24. LTDL (2007) The Legacy Tobacco Document Library (LTDL). University of California, San Francisco
25. Maltoni D, Maio D, Jain A, Prabhakar S (2009) Handbook of fingerprint recognition. Springer Science & Business Media
26. Matsumoto M, Nishimura T (1998) Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation (TOMACS) 8(1):3–30
27. Otsu N (1979) A threshold selection method from gray-level histograms. IEEE transactions on systems, man, and cybernetics 9(1):62–66
28. Sauvola J, Pietikäinen M (2000) Adaptive document image binarization. Pattern recognition 33(2):225–236
29. Sutheebanjard P, Premchaiswadi W (2011) Efficient scan mask techniques for connected components labeling algorithm. EURASIP Journal on image and Video Processing 2011(1):1–20
30. Tarjan RE (1975) Efficiency of a good but not linear set union algorithm. Journal of the ACM 22(2):215–225
31. Torralba A, Efros AA (2011) Unbiased Look at Dataset Bias. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, pp 1521–1528
32. Wu K, Otoo E, Suzuki K (2005) Two Strategies to Speed up Connected Component Labeling Algorithms. Tech. Rep. LBNL-59102, Lawrence Berkeley National Laboratory
33. Wu K, Otoo E, Suzuki K (2009) Optimizing two-pass connected-component labeling algorithms. Pattern Analysis and Applications 12(2):117–135
34. Zhao H, Fan Y, Zhang T, Sang H (2010) Stripe-based connected components labelling. Electronics letters 46(21):1434–1436

Table 4 Average run-time results in ms obtained under both Linux (a) and Windows (b) with GCC 5.1.0 compiler. The bold values represent the best labels solver for a specific CCL algorithm and dataset, the red ones point out the best algorithm for a given dataset.

		3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
(a) *Linux with GCC 5.1.0*	SAUF_RemSP	**1.123**	0.404	6.832	**3.036**	**0.556**	**11.209**	**41.031**
	SAUF_TTA	1.136	0.424	6.960	3.178	0.573	11.243	41.719
	SAUF_UFPC	1.143	0.432	6.998	3.083	0.570	11.358	42.042
	SAUF_UF	1.126	**0.400**	**6.827**	3.065	0.559	11.230	41.438
	BBDT_RemSP	0.687	**0.330**	**4.376**	**1.985**	**0.391**	**6.477**	**24.632**
	BBDT_TTA	0.699	0.339	4.512	2.047	0.398	6.612	25.561
	BBDT_UFPC	0.688	0.335	4.480	2.003	0.395	6.571	25.223
	BBDT_UF	**0.682**	0.333	4.412	2.037	0.395	6.500	24.882
	CCIT_RemSP	0.801	0.378	4.915	2.303	**0.470**	7.447	27.100
	CCIT_TTA	**0.759**	**0.367**	**4.747**	**2.267**	0.471	**6.975**	**26.648**
	CCIT_UFPC	0.839	0.412	5.369	2.503	0.518	7.945	30.316
	CCIT_UF	0.853	0.403	5.386	2.465	0.504	8.035	30.149
	CTB_RemSP	0.669	**0.316**	**4.149**	2.097	0.458	**6.218**	24.501
	CTB_TTA	0.675	0.317	4.175	**2.076**	**0.443**	6.219	24.576
	CTB_UFPC	**0.669**	0.317	4.163	2.091	0.456	6.254	**24.452**
	CTB_UF	0.673	0.325	4.194	2.078	0.452	6.263	24.537
	PRED_RemSP	0.689	0.323	4.254	**1.972**	0.414	6.310	24.579
	PRED_TTA	0.696	0.328	4.291	1.993	0.416	6.354	25.000
	PRED_UFPC	0.691	0.330	4.289	2.003	0.422	6.361	24.853
	PRED_UF	**0.684**	**0.323**	**4.219**	1.974	**0.413**	**6.259**	**24.469**
	LSL_STD_TTA	2.100	0.689	11.640	5.638	0.949	18.400	64.207
	LSL_STD_UF	**2.095**	**0.685**	**11.608**	**5.504**	**0.944**	**18.337**	**64.048**
	LSL_STDZ_TTA	1.814	0.595	9.667	4.760	0.823	15.049	52.201
	LSL_STDZ_UF	**1.796**	**0.583**	**9.511**	**4.572**	**0.811**	**14.794**	**51.313**
	LSL_RLE_TTA	1.714	0.655	9.380	4.587	0.837	13.966	49.483
	LSL_RLE_UF	**1.706**	**0.646**	**9.330**	**4.435**	**0.829**	**13.887**	**49.240**
	LSL_RLEZ_TTA	1.713	0.655	9.385	4.587	0.837	13.983	49.543
	LSL_RLEZ_UF	**1.696**	**0.637**	**9.221**	**4.396**	**0.818**	**13.785**	**48.687**
	DiStefano	1.198	0.629	8.377	4.232	0.857	12.753	58.009
	CT	1.435	1.027	11.447	5.371	1.073	15.804	66.645
	SBLA	1.339	0.685	9.655	4.454	0.821	14.406	56.906
	NULL	0.395	0.094	1.791	0.849	0.160	2.834	9.685
(b) *Windows with GCC 5.1.0*	SAUF_RemSP	1.123	0.420	7.932	3.476	**0.578**	13.293	63.635
	SAUF_TTA	1.139	0.429	7.945	3.525	0.590	**13.264**	64.126
	SAUF_UFPC	1.141	0.442	8.029	3.485	0.589	13.425	**63.276**
	SAUF_UF	**1.121**	**0.414**	**7.863**	**3.476**	0.581	13.265	63.607
	BBDT_RemSP	0.723	0.362	5.490	2.311	0.422	8.756	42.290
	BBDT_TTA	0.733	0.365	5.505	2.317	0.423	8.765	42.601
	BBDT_UFPC	0.724	0.365	5.500	2.315	0.423	8.760	42.363
	BBDT_UF	**0.609**	**0.321**	**4.715**	**1.977**	**0.370**	**7.484**	**36.727**
	CCIT_RemSP	**0.845**	**0.400**	**6.307**	**2.806**	**0.508**	**10.126**	**50.326**
	CCIT_TTA	0.894	0.418	6.665	3.009	0.540	10.712	53.974
	CCIT_UFPC	0.855	0.424	6.431	2.899	0.545	10.196	51.185
	CCIT_UF	0.853	0.416	6.389	2.851	0.529	10.157	50.337
	CTB_RemSP	0.674	0.347	5.306	2.614	0.509	8.380	48.303
	CTB_TTA	0.687	0.349	5.349	2.637	0.514	8.440	48.785
	CTB_UFPC	**0.659**	0.346	5.208	2.606	0.516	**8.215**	47.846
	CTB_UF	0.675	**0.335**	**5.203**	**2.436**	**0.468**	8.260	**44.461**
	PRED_RemSP	0.693	0.349	5.363	2.411	0.446	8.436	43.561
	PRED_TTA	0.707	0.355	5.431	2.436	0.450	8.505	44.071
	PRED_UFPC	0.696	0.353	5.382	2.422	0.453	8.447	43.708
	PRED_UF	**0.686**	**0.332**	**5.247**	**2.355**	**0.431**	**8.335**	**42.662**
	LSL_STD_TTA	2.026	0.658	15.620	6.761	0.946	26.373	119.099
	LSL_STD_UF	**2.011**	**0.653**	**15.567**	**6.743**	**0.940**	**26.299**	**118.757**
	LSL_STDZ_TTA	1.745	0.565	13.590	5.867	0.823	23.043	104.320
	LSL_STDZ_UF	**1.710**	**0.552**	**13.402**	**5.784**	**0.808**	**22.753**	**102.928**
	LSL_RLE_TTA	1.635	**0.622**	**13.279**	**5.667**	**0.832**	**21.892**	**99.882**
	LSL_RLE_UF	**1.627**	0.627	13.311	5.734	0.835	21.918	100.168
	LSL_RLEZ_TTA	1.635	0.628	13.314	5.736	0.836	21.917	100.093
	LSL_RLEZ_UF	**1.610**	**0.601**	**13.067**	**5.584**	**0.811**	**21.687**	**98.852**
	DiStefano	0.820	0.555	6.879	3.619	0.738	10.293	72.852
	CT	1.431	1.037	12.332	5.716	1.099	17.953	98.586
	SBLA	1.263	0.682	10.089	4.569	0.817	15.494	80.206
	NULL	0.364	0.096	2.452	1.086	0.163	4.397	21.378