Two More Strategies to Speed Up Connected Components Labeling Algorithms

Federico Bolelli, Michele Cancilla, Costantino Grana

Dipartimento di Ingegneria "Enzo Ferrari"
Università degli Studi di Modena e Reggio Emilia
Via Vivarelli 10, Modena MO 41125, Italy
name.surname@unimore.it

Abstract. This paper presents two strategies that can be used to improve the speed of Connected Components Labeling algorithms. The first one operates on optimal decision trees considering image patterns occurrences, while the second one articulates how two scan algorithms can be parallelized using multi-threading. Experimental results demonstrate that the proposed methodologies reduce the total execution time of state-of-the-art two scan algorithms.

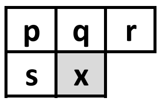
Keywords: Connected Components Labeling; Binary Decision Trees; Parallelization; Optimization.

1 Introduction

Connected Components Labeling (CCL) of binary image is a fundamental task in several image processing and computer vision applications ranging from video surveillance to medical imaging. CCL transforms a binary image into a symbolic one in which all pixels belonging to the same connected component are given the same label. Thus, this transformation is required whenever a computer program needs to identify independent components. Moreover, given that labeling is the base step of most real time applications it is required to be as fast as possible. Since labeling is a well-defined problem and the exact solution for a given image should be provided as output, the proposals of the last twenty years have focused on performance optimization. A significant improvement was given by the introduction of the *Union-Find* [17] approach for label equivalences resolution and array-based data structures [11].

Most of the labeling algorithms employ a raster scan mask to look at neighborhood of a pixel and to determine its correct label. As shown in literature, the decision table associated to the mask which rules the scan step can be converted to an optimal binary decision tree by the use of a dynamic programming approach [8]. This approach leads to a reduction of total memory accesses and total execution time.

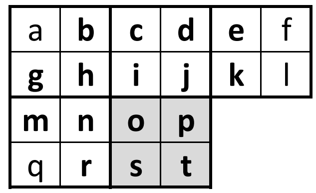
Another strategy to improve the performance of existing algorithms could be the parallelization. A simple process which divides the input image into horizontal stripes and computes labeling separately on each one is described in [3]. The



(a)

							assign				merge	
x	p	q	r	s	**no action**	**new label**	x = p	x = q	x = r	x = s	x = p + r	x = r + s
0	-	-	-	-	**1**							
1	0	0	0	0		**1**						
1	1	0	0	0			**1**					
1	0	1	0	0				**1**				
1	0	0	1	0					**1**			
1	0	0	0	1						**1**		
1	1	1	0	0			*1*	**1**				
1	1	0	1	0							**1**	
1	1	0	0	1			*1*			**1**		
1	0	1	1	0				**1**	*1*			
1	0	1	0	1				**1**		*1*		
1	0	0	1	1								**1**
1	1	1	1	0			*1*	**1**	*1*			
1	1	1	0	1			*1*	**1**		*1*		
1	1	0	1	1							*1*	**1**
1	0	1	1	1				**1**	*1*	*1*		
1	1	1	1	1			*1*	**1**	*1*	*1*		

(b)



(c)

Fig. 1: (a) Masks used to compute labels of pixel x by SAUF algorithm, (b) its associated *OR*-decision table. Finally, (c) is the mask used to compute label of pixels o, p, s, and t by the BBDT algorithm.

general problem of Connected Components Labeling on parallel architecture was exhaustively threated in a theoretical way by [1].

Recently, our research group released an open source C++ framework for performance evaluation of CCL algorithms. The benchmarking system called YACCLAB [7] (acronym for *Yet Another Connected Components Labeling Benchmark*) collects, runs, and tests the state-of-the-art labeling algorithms over an extremely variety of different datasets solving the problem of fair evaluation of different strategies. Results shown in this paper are produced using this tool.

With this work we experiment two strategies to improve the performance of Connected Components Labeling applying them to the SAUF (*Scan Array Union Find*) algorithm proposed by Wu *et al.* [18] and BBDT (*Block Based with Decision Tree*) strategy proposed by Grana *et al.* [8]. Firstly, we have transformed the decision trees used by both the algorithms considering the occurrence probability of each pattern in a mask and on a reference dataset. Secondly, we have parallelized both the algorithms, evaluating the benefits and limits of different approaches. Then we have tested all resulting algorithms on different datasets and environments to highlights their different behaviors.

The rest of this paper is organized as follows: in Section 2 we give an overview of existing Connected Components Labeling algorithms; Sections 3 and 4 contain the description of the implemented strategies, which are then evaluated in Section 5. Finally, we draw the conclusions in Section 6.

2 Previous Works

Connected Components Labeling has a very long story full of different strategies that can be classified in three different main groups:

– *Raster Scan* algorithms which scan the image exploiting a mask (see for instance Figure 1) and solve equivalences between labels using different strategies. *Multiscans* approaches [10], for example, scan the image alternatively in forward and background directions to propagate labels until no changes occur in the output matrix. On the other hand, modern *Two Scan* algorithms [8,12,17,18] solve equivalences between labels on-line during the first scan, usually storing them in a Union-Find tree (*i.e.* a 1-D array also called P). Provisional labels in the output image are then replaced with the smallest equivalent label found in the flattened P array.

- *Searching and Label Propagation* algorithms [15] scan the image until an unlabeled pixel is found: it receives a new label which is then repetitively propagated to all connected pixels. The process end when unlabeled pixels no longer exist. These algorithms scan the image in an irregular way.
- *Contour Tracing* techniques [4] exploit a single raster scan over the image. During this process all pixels in both the contour and the immediately external background of an object are clockwise tagged in a single operation. Finally, the connected components have to be filled propagating contours' labels.

Two scan algorithms have revealed best performances [7], so our analysis focuses on them.

The SAUF algorithm [17,18] implements the Union-Find technique with path compression and exploits a decision tree for accessing only the minimum number of already labeled pixels.

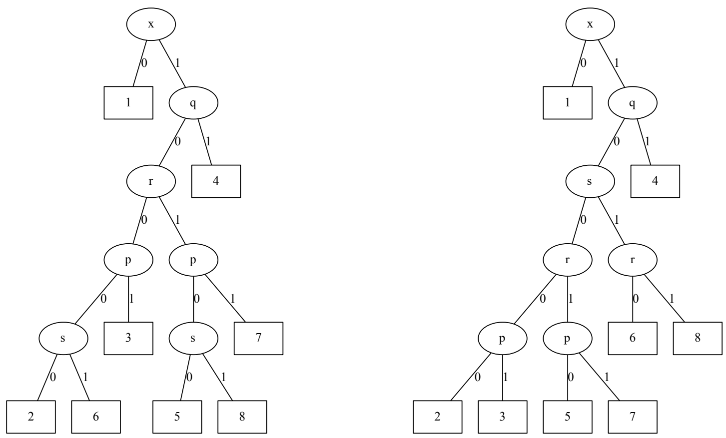
In [8] it was proven that, when performing 8-connected labeling, the scanning process can be extended to block-based scanning, that scans the image in 2×2 blocks (BBDT). In that case, because of the large number of possible combinations, the final decision tree is generated automatically by another program [9].

He *et al.* [12] recently observed that BBDT checks many pixels repeatedly, because after labeling one pixel, the mask moves to the next one, but many pixels in the current mask are overlapped to the previous ones, which may have already been checked. They thus proposed a *Configuration Transition Based* (CTB) algorithm which introduces different configurations *states* in order to make further decisions. This procedure reduces the number of pixels checked and thus speeds up the labeling procedure.

Finally, Grana *et al.* [6] proposed an approach called *Optimized Connected Component Labeling with Pixel Prediction* (PRED) which employs a reproducible strategy able to avoid repeatedly checking the same pixels multiple times. The first scan phase of PRED is ruled by a forest of decision trees connected into a single graph. Each tree derives from a reduction of one complete optimal decision tree.

3 Pattern Analysis and Modeling of Decision Trees

In [17] Wu *et al.* have shown that when considering 8-connected components in a 2D image it is advantageous to exploit the dependencies between pixels in the scan mask to reduce memory accesses. This can be performed with the use of a decision tree which can be derived from the decision table (Figure 1b) associated



(a) (b)

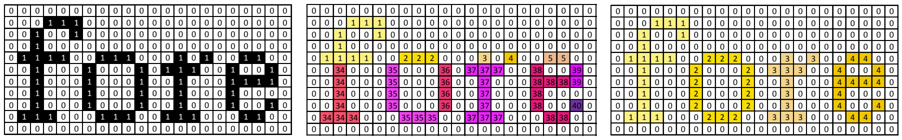
Fig. 2: Two of the optimal decision trees derivable from Rosenfeld's mask (Figure 1a). Nodes (circle box) show the conditions to check and leaves (square box) contain the action to perform: (1) nothing to do, (2) new label, (3) $x = p$, (4) $x = q$, (5) $x = r$, (6) $x = s$, (7) $x = r + p$ and (8) $x = r + s$.

to the scan mask. In [8] Grana *et al.* proposed an automatic strategy to convert *AND*-decision tables to *OR*-decision tables and produce the associated optimal decision tree by the use of a dynamic programming technique originally described by Schumacher *et al.* [16].

The basic concept behind the creation of a simplified tree from a decision table (with n conditions) is that if two branches lead to the same action the condition from which they originate may be removed. Thus, this conversion can be interpreted as the partitioning of an n-dimensional hypercube where the vertexes correspond to the 2^n possible rules. Associating to each condition removal a unitary gain we can select the tree which maximizes the total gain and minimizes the number of memory accesses. Moreover, in [9], an exhaustive search procedure is provided to select the most convenient action among the alternatives in the *OR*-decision table. This strategy is also proven to generate always one of the possible optimal decision trees. A more detailed description of the tree generation can be found in [8].

In Figure 2 we have reported an example of two equivalent optimal decision tree associated to the SAUF algorithm. The Figure 2a shows the tree presented by Wu *et al.* in [17], while 2b is another tree obtainable from the original *OR*-decision table.

The novelty of the proposed approach lies on the analysis of the occurrence frequencies of each pattern derived from a mask on a reference dataset. If we generate the trees considering pattern probabilities, we are able to reduce the



(a) (b) (c)

Fig. 3: (a) Example of binary image depicting text, (b) its labeling considering only the first scan of the parallel implementation of SAUF (two threads), and finally (c) its labeling result.

average number of memory accesses for a specific kind of images. In this case the gain obtained by a condition removal is related to the frequency of associated patterns. We expect that the greater the complexity of the mask is, the higher the gain given by this operation will be.

4 Parallel Connected Components Labeling

Modern computer architectures are multi-cores and support multi-threaded applications, so it is convenient to analyze how much the performance of CCL algorithms scale up when multi-threading is involved. The goal is to spread the computational cost among different threads without increasing the execution time with heavy synchronization mechanisms. A basic approach to parallelize two scan labeling algorithms divides images into stripes (chunks) and operates with the following three steps [3]:

1. Compute first scan on each stripe (in parallel);
2. Merge border labels and flatten equivalences array P;
3. Compute second scan on each stripe (in parallel).

Assuming to have n workers available to compute labeling, we can spread the computational cost for the first scan between them dividing the image in n parts. In order to be cache friendly we choose to cut the image horizontally. Each stripe is then labeled independently (first scan) by each worker using provisional labels.

To avoid getting overlapped labels among different stripes, each thread must use a different set of initial provisional labels. When using 8-connectivity, each 2×2 block of pixels can only contain a single label, so the starting label number for a given thread can be easily calculated as

$$\left\lfloor \frac{r_i + 1}{2} \right\rfloor \cdot \left\lfloor \frac{w + 1}{2} \right\rfloor + 1, \tag{1}$$

where r_i represents the index of the i-th chunk's first row and w is the image width. The background will always take label zero. During the first scan, threads have also to collect any possible equivalence between labels and store it in the

P array. To avoid collisions each stripe deals with a disjoint set of labels and this implies non overlapped accesses to the array storing the equivalences. In Figure 3b an example of the resulting image from the first scan is reported when two threads are involved.

After the first scan we have to establish a bridge between two adjacent chunks with the operation of Merge (Union of the Union-Find approach). This is required since chunks are labeled with uncorrelated provisional labels. In order to compute Merge we use masks reported in Figure 4.

There are basically two strategies to figure out this problem and they are listed below.

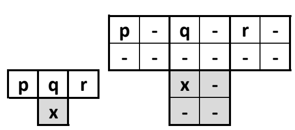
- *Sequential Merge* scans each chunk's border sequentially and does not employ multi-threading, avoiding collision management in label equivalences array.
- *Logarithmic Merge* already proposed in [3], employs multi-threading to speed up this operation. Indeed, we can operate concurrently on two or more chunks when they are not accessed at the same time by another thread.

For example, if we consider an image processed with eight threads hence divided into eight stripes the Sequential Merge requires 7 stages for merging chunks while the Logarithmic one only 3 ($\log_2 8$).

The second approach, compared to the Sequential one, leads, at least in theory, to better performance. However, the management of Logarithmic Merge is more complex when the number of threads is not a power of two, so a correct implementation of it must include additional conditional statements. Hence, as shown in Table 1 the Sequential Merge operation performs better if compared with Logarithmic. Due to this, all parallel results shown in Section 5 are obtained using Sequential Merge.

It is important to notice that the described approach leads to a fragmented array of equivalences and consequently the Union-Find tree needs to be flattened in a hop-by-hop way. This behavior is linked to the possibility of having a lot of unused provisional labels associated to each chunk in the P array.

To conclude the labeling process, the second scan is done in parallel by the same number of threads and using the same mask of the first scan. Second scan better suits the multi-threading approach because it contains few conditional statements.



(a) (b)

Fig. 4: (a) Merge mask used by SAUF parallel algorithm and (b) by BBDT parallel algorithm.

Table 1: Comparison between Sequential and Logarithmic Merge incidence (in percentage) with respect to the total execution time. The reported values are calculated considering all YACCLAB datasets, with different number of threads (hence chunks) and using BBDT parallel algorithm.

	2 chunks	*4 chunks*	*8 chunks*	*16 chunks*	*24 chunks*
Sequential	0.45 %	1.49 %	3.35 %	5.34 %	7.10 %
Logarithmic	0.84 %	3.64 %	6.11 %	7.63 %	8.73 %

5 Experimental Evaluation

In order to produce an overall view of the proposed methods performance, we ran each algorithm using the YACCLAB tool on different environments: a Windows server with two Intel Xeon X5650 *CPU* @ 2.67GHz and Microsoft Visual Studio 2013 and a Windows PC with an Intel i5-6600 *CPU* @ 3.30GHz running Microsoft Visual Studio 2013. All tests were repeated 10 times, and for each image the minimum execution time was considered in order to reduce the effects of background processes.

In the following of this section, we use acronyms to refer to the available algorithms: BBDT is the Block Based with Decision Trees algorithm by Grana *et al.* [8], BBDT_ALL and BBDT_ONE are the versions of the BBDT algorithm with optimal trees generated considering respectively patterns frequency of all datasets described in Section 5.1 and patterns frequency of the same dataset on which the algorithm is tested. BBDT_$\langle n \rangle$ is the parallel implementation of the BBDT algorithm run with n threads, SAUF is the Scan Array Union Find algorithm by Wu *et al.* [18], SAUF_ALL refers to the algorithm that uses the optimal tree generated considering patterns frequency on all YACCLAB datasets, and, finally, SAUF_$\langle n \rangle$ is the SAUF parallel version run using n threads. BBDT and SAUF are the algorithms currently included in the OpenCV's *connectedComponents* function.

5.1 Datasets

The datasets used for tests are currently included in YACCLAB. All images are provided in 1 bit per pixel PNG format, with 0 (black) being background and 1 (white) being foreground. The images can be grouped by their nature as follows:

- *MIRflickr* is composed by natural images, it contains 25,000 standard resolution images taken from Flickr, with an average resolution of 0.18 megapixels.
- *Hamlet* and *Tobacco* [13] are two set of document images. The first one contains 104 images scanned from a version of the Hamlet found on Project Gutenberg. The second one is composed of 1290 document images and it is a realistic database for document image analysis research.
- *Medical* dataset, provided by Dong *et al.* [5], is composed of histological images and allow to cover this fundamental field.

Table 2: Results in ms on Windows PC with i5-6600 @ 3.30 GHz and Microsoft Visual Studio 2013 (lower is better).

	BBDT	BBDT_ALL	BBDT_ONE
3DPeS	0.678	**0.621**	0.645
Fingerprints	0.343	0.322	**0.320**
Hamlet	5.284	**5.048**	5.252
Medical	2.273	**2.154**	2.213
MIRflickr	0.450	**0.414**	0.424
Tobacco800	7.752	**7.283**	7.431

– *Fingerprints* [14] accommodates 960 fingerprint images collected by using low-cost optical sensors or synthetically generated.
– *3DPeS* [2] is a surveillance dataset mainly designed for people re-identification in multi-camera systems with non-overlapped fields of view. Images have an average amount of 0.41 million of pixels to analyze and 320 components to label.
– *Random* contains black and white random noise images with nine different foreground densities (from 10% up to 90%).

A more detailed description of the YACCLAB's datasets can be found in [7].

5.2 Frequencies Results

Firstly, we have run the tree generator algorithm described in Section 3 on the SAUF mask considering the patterns frequency of all YACCLAB datasets. The resulting tree is the one reported in Figure 2b, that is different from the one proposed by Wu *et al.* (Figure 2a), but it is one of the optimal equivalent trees derivable from decision table in Figure 1b without considering frequencies. Indeed, the small number of conditions for this case limits the number of possible operations on trees. However, Table 3b shows that the tree generated by our algorithm requires from 0.03% to 0.27% less memory accesses when applied on real datasets. Instead, as expected, on random dataset the number of accesses is almost the same due to its uniform distribution of patterns' probability. In Table 3a comparison between execution time is also reported.

The BBDT algorithm, instead, requires much more complex trees because of the high number of conditions to check and thus leaves space for better optimization. In Table 2 the execution times of three BBDT algorithms which use different trees are compared: the first one is obtained with uniform frequencies (BBDT), the second one is generated considering all YACCLAB dataset frequencies (BBDT_ALL), and, finally, the third one employs the tree generated by the frequencies of dataset on which it is tested (BBDT_ONE). Table 2 demonstrates that the pattern analysis leads to better performance compared to the original BBDT algorithm. It is important to highlight that the best results are obtained with BBDT_ALL algorithm instead of BBDT_ONEs: this is an unexpected result considering how trees have been generated. From our knowledge

Table 3: (a) Results in ms on Windows with Intel i5-6600 @ 3.30GHz and Microsoft Visual Studio 2013 (lower is better) with optimization disabled. (b) Analysis of memory accesses performed on YACCLAB dataset.

	SAUF	SAUF_ALL
3DPeS	2.059	2.037
Fingerprints	0.847	0.844
Hamlet	15.070	14.834
Medical	6.505	6.401
MIRflickr	1.053	1.059
Tobacco800	24.667	24.606
Random	28.731	29.065

(a)

	SAUF	SAUF_ALL
3DPeS	2 069 566	2 069 002
Fingerprints	809 288	807 136
Hamlet	14 325 113	14 310 510
Medical	7 035 408	7 028 268
MIRflickr	1 170 066	1 168 648
Tobacco800	23 873 949	23 857 166
Random	20 963 033	20 963 038

(b)

the only reasonable explanation is related to the code optimizations applied by the compiler.

The improvements related to frequency analysis, albeit limited, are significant given the maturity of the problem and the performance achieved by the best existing algorithms.

5.3 Parallel Results

The parallel implementation of both BBDT and SAUF is based on the OpenCV's build in $parallel\_for\_$ function. This function runs the parallel loop of one of the available parallel frameworks, selecting it at compilation time. All parallel results presented in this paper are obtained using *Intel Threading Building Blocks* (TBB). Table 4 shows the average execution time of sequential and parallel version of both the algorithms on all dataset and with different number of threads. Experiments reveal that the overhead introduced by the $parallel\_for\_$ with TBB is negligible *i.e.* the execution time of a sequential algorithm and its parallel version implemented with OpenCV parallel framework and tested with one thread is the same. As shown in Table 4, the speed up obtained with two threads on SAUF is $\times 1.5$ in average and it increases up to $\times 4$ on random dataset when 12 threads are involved. Starting from 16 threads (*i.e.* when hyper-threading is involved and threads share cache of both first and second level) the performance of SAUF decrease. This could be explained by the increment of instruction cache misses due to data overflow. For what concerns BBDT algorithm, experimental results demonstrate a greater speed up with low number of threads (*i.e.* $\times 1.7$ with 2 threads, up to $\times 4.7$ on random dataset with 8 threads). However, increasing the parallelism, as it happens with the SAUF algorithm, leads to worse performance. Once again this behavior could be related to the increase of instruction cache misses that, in this case, occur starting from 8 threads, because BBDT code footprint is much bigger than SAUF one.

Table 4: Average results in ms on a virtual Windows workstation with two Intel Xeon X5650 *CPU* @ 2.67GHz (6 physical cores and 12 logical processors per socket) and Microsoft Visual Studio 2013 (lower is better).

	SAUF	SAUF_2	SAUF_4	SAUF_8	SAUF_12	SAUF_16	SAUF_24
3DPeS	1.817	1.258	1.013	0.886	**0.845**	0.972	0.969
Fingerprints	0.793	0.548	0.431	0.361	**0.338**	0.402	0.426
Hamlet	13.449	8.682	6.726	5.651	**5.338**	5.615	5.446
Medical	6.316	4.414	3.104	2.626	**2.542**	2.742	2.745
MIRflickr	1.053	0.770	0.608	0.544	**0.543**	0.618	0.657
Tobacco800	22.075	14.362	11.083	9.445	**9.057**	9.457	9.166
Random	31.525	19.554	11.956	8.695	**7.795**	9.144	8.605

	BBDT	BBDT_2	BBDT_4	BBDT_8	BBDT_12	BBDT_16	BBDT_24
3DPeS	1.274	0.794	**0.609**	0.720	0.812	0.918	1.002
Fingerprints	0.606	0.386	0.298	**0.274**	0.290	0.347	0.388
Hamlet	10.528	5.989	**4.342**	4.528	5.186	5.423	6.146
Medical	4.945	3.051	**1.831**	2.060	2.407	2.621	2.998
MIRflickr	0.790	0.520	**0.393**	0.438	0.479	0.559	0.622
Tobacco800	16.587	9.590	**6.518**	7.207	8.206	9.215	10.375
Random	25.106	13.177	7.084	**5.375**	6.004	7.567	8.387

6 Conclusions

In this paper we presented two different approaches able to speed up two scan Connected Components Labeling algorithms. The first strategy operates on optimal decision trees in order to reduce memory accesses and improve performance. This transformation processes can be performed off-line and requires to know the occurrences of all possible patterns on a reference dataset or a group of them. As shown, the proposed approach leads to an improvement of the performance of CCL if applied to complex decision trees, such as the one implemented by BBDT. The second strategy employs multi-threading in order to spread computational cost among different processes. Differently from the first approach it is easily applicable to all two scan algorithms. The source code of the described algorithms is included in YACCLAB: we strongly believe that given the maturity of the problem and the subtlety involved in the implementation, it should be mandatory to allow the community to reproduce the results without forcing everyone to reimplement every proposal. To conclude, the parallel versions of BBDT and SAUF algorithms discussed in this paper have been submitted to OpenCV.

References

1. Alnuweiri, H.M., Prasanna, V.K.: Parallel architectures and algorithms for image component labeling. IEEE Transactions on Pattern Analysis and Machine Intelligence 14(10), 1014–1034 (1992)
2. Baltieri, D., Vezzani, R., Cucchiara, R.: 3DPeS: 3D People Dataset for Surveillance and Forensics. In: Proceedings of the 2011 joint ACM workshop on Human gesture and behavior understanding. pp. 59–64. ACM (2011)
3. Cabaret, L., Lacassagne, L., Etiemble, D.: Parallel light speed labeling: An efficient connected component labeling algorithm for multi-core processors. In: Image Processing (ICIP), 2015 IEEE International Conference on. pp. 3486–3489. IEEE (2015)
4. Chang, F., Chen, C.J., Lu, C.J.: A linear-time component-labeling algorithm using contour tracing technique. Computer Vision and Image Understanding 93(2), 206–220 (2004)
5. Dong, F., Irshad, H., Oh, E.Y., Lerwill, M.F., Brachtel, E.F., Jones, N.C., Knoblauch, N.W., Montaser-Kouhsari, L., Johnson, N.B., Rao, L.K., et al.: Computational Pathology to Discriminate Benign from Malignant Intraductal Proliferations of the Breast. PloS one 9(12), e114885 (2014)
6. Grana, C., Baraldi, L., Bolelli, F.: Optimized connected components labeling with pixel prediction. In: International Conference on Advanced Concepts for Intelligent Vision Systems. pp. 431–440. Springer (2016)
7. Grana, C., Bolelli, F., Baraldi, L., Vezzani, R.: YACCLAB - Yet Another Connected Components Labeling Benchmark. In: 23rd International Conference on Pattern Recognition. ICPR (2016)
8. Grana, C., Borghesani, D., Cucchiara, R.: Optimized Block-based Connected Components Labeling with Decision Trees. IEEE Transactions on Image Processing 19(6), 1596–1609 (2010)
9. Grana, C., Montangero, M., Borghesani, D.: Optimal decision trees for local image processing algorithms. Pattern Recognition Letters 33(16), 2302–2310 (2012)
10. Haralick, R.: Some neighborhood operators. In: Real-Time Parallel Computing, pp. 11–35. Springer (1981)
11. He, L., Chao, Y., Suzuki, K.: A Linear-Time Two-Scan Labeling Algorithm. In: International Conference on Image Processing. vol. 5, pp. 241–244 (2007)
12. He, L., Zhao, X., Chao, Y., Suzuki, K.: Configuration-Transition-Based Connected-Component Labeling. IEEE Transactions on Image Processing 23(2), 943–951 (2014)
13. Lewis, D., Agam, G., Argamon, S., Frieder, O., Grossman, D., J.Heard: Building a test collection for complex document information processing. In: Proc. 29th Annual Int. ACM SIGIR Conference. pp. 665–666 (2006)
14. Maltoni, D., Maio, D., Jain, A., Prabhakar, S.: Handbook of fingerprint recognition. Springer Science & Business Media (2009)
15. Rosenfeld, A., Kak, A.: Digital picture processing. No. v. 1 in Computer science and applied mathematics, Academic Press (1982)
16. Schumacher, H., Sevcik, K.C.: The synthetic approach to decision table conversion. Commun. ACM 19(6), 343–351 (Jun 1976)
17. Wu, K., Otoo, E., Suzuki, K.: Two Strategies to Speed up Connected Component Labeling Algorithms. Tech. Rep. LBNL-59102, Lawrence Berkeley National Laboratory (2005)
18. Wu, K., Otoo, E., Suzuki, K.: Optimizing two-pass connected-component labeling algorithms. Pattern Analysis and Applications 12(2), 117–135 (2009)