This is the peer reviewd version of the followng article:

Spaghetti Labeling: Directed Acyclic Graphs for Block-Based Connected Components Labeling / Bolelli, Federico; Allegretti, Stefano; Baraldi, Lorenzo; Grana, Costantino. - In: IEEE TRANSACTIONS ON IMAGE PROCESSING. - ISSN 1057-7149. - 29:1(2020), pp. 1999-2012. [10.1109/TIP.2019.2946979]

Terms of use:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

12/10/2024 08:28

(Article begins on next page)

Spaghetti Labeling: Directed Acyclic Graphs for Block-Based Connected Components Labeling

Federico Bolelli, *Student Member, IEEE*, Stefano Allegretti, Lorenzo Baraldi, and Costantino Grana, *Member, IEEE*

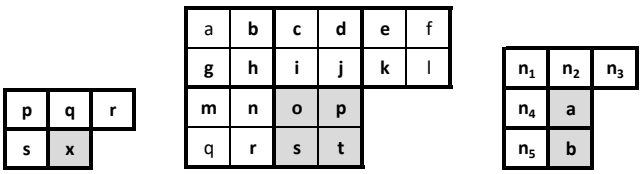
***Abstract*—Connected Components Labeling is an essential step of many Image Processing and Computer Vision tasks. Since the first proposal of a labeling algorithm, which dates back to the sixties, many approaches have optimized the computational load needed to label an image. In particular, the use of decision forests and state prediction have recently appeared as valuable strategies to improve performance. However, due to the overhead of the manual construction of prediction states and the size of the resulting machine code, the application of these strategies has been restricted to small masks, thus ignoring the benefit of using a block-based approach. In this paper, we combine a block-based mask with state prediction and code compression: the resulting algorithm is modeled as a Directed Rooted Acyclic Graph with multiple entry points, which is automatically generated without manual intervention. When tested on synthetic and real datasets, in comparison with optimized implementations of state-of-the-art algorithms, the proposed approach shows superior performance, surpassing the results obtained by all compared approaches in all settings.**

***Index Terms*—Connected Components Labeling, Optimal Decision Trees, Direct Acyclic Graphs, Image Processing**

I. Introduction

CONNECTED Components Labeling (CCL) is a fundamental image processing algorithm that transforms an input binary image into a symbolic one in which all pixels of the same connected component (object) are given the same label. Introduced by Rosenfeld and Pfaltz [1], sequential CCL on binary images has been in use for more than 50 years in multiple image processing and computer vision tasks, including Object Tracking [2], Video Surveillance [3], Image Segmentation [4], [5], Medical Imaging Applications [6], [7], [8], [9], Document Restoration [10], [11], Graph Analysis [12], [13], and Environmental Applications [14].

The CCL problem has a unique and exact solution. Different algorithms can be used to obtain the symbolic image, which will always have the same content, except for the specific label assigned to each connected component. The only difference is thus the time required to obtain the result.



(a) Rosenfeld (b) Grana (c) He

Fig. 1. Example of scan masks. Gray squares identify current pixels to be labeled using information extracted from white pixels.

Since 1966, many papers showed algorithms to improve the efficiency of CCL and, given the relevance of the task, this still represents a hot research topic [15], [16], [17], [18], [19], [20], [21]. The first significant improvement has been provided by Wu *et al.* [22], who proved an optimal strategy to reduce the average number of load/store operations during the scan of the input image, driven by Rosenfeld mask (Fig. 1a). They exploited a manually identified Decision Tree (DTree) to minimize the number of neighboring pixels to be visited in order to evaluate the label of the current one. This algorithm has been named SAUF (Scan Array-based Union Find). Grana *et al.* [23] subsequently introduced a major breakthrough, consisting in a 2×2 block-based approach (Fig. 1b), which modeled the CCL problem as a decision problem, and applied decision tables and trees to automatically generate the algorithm source code. They named this algorithm BBDT (Block-Based with Decison Trees).

Many improvements have been proposed since then [20], and few of them introduced significantly novel ideas, in particular:

- a proved algorithm to produce optimal decision trees [24];
- realizing that it is possible to use a finite state machine to summarize the value of pixels already inspected by the horizontally moving scan mask [16];
- combining decision trees and configuration transitions in a decision forest, in which each previous pattern allows to "predict" some of the current configuration pixels values, thus allowing automatic code generation [25];
- switching from decision trees to Directed Rooted Acyclic Graphs (DRAGs), to reduce the machine code footprint and lessen its impact on the instruction cache [26].

Prediction, as introduced by He *et al.* [27] and later improved in [16] and [28], has proven to be one of the most useful additions, as it allows to exploit already available information, save expensive load/store operations, and reduce execution time consequently. The idea behind prediction is tied to the fact that most existing algorithms scan the image and look at the neighborhood of a pixel through a mask. For each step of the scan process, an action is performed depending on the values of pixels inside the mask. When the mask is shifted along a row of the image it always contains some of the pixels it already contained in the previous step, though in different locations. If those pixels were indeed checked in the previous mask step, a second read of their value can be avoided by their removal from the decision process.

Anyway, in [16] the mask used was smaller than the one

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

Fig. 2. *AND*-decision table for Rosenfeld mask. A different action for each condition outcome (mask configuration) is provided. To produce a more compact visualization the redundant logic have been reduced by means of the indifferent condition (represented by "–"). A condition marked with "–" does not affect the decision.

							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*		

Fig. 3. *OR*-decision table for Rosenfeld mask.

used in BBDT, because it was unfeasible to manually analyze all possible combinations produced by the prediction states. On the other hand, the procedure proposed in [25] is suitable to be automatized, but still a small mask was employed. The reason, in this case, is that the larger the mask is, the more decision trees will populate the resulting forest, and the higher every tree will be. The machine code that implements the algorithm resulting from the application of prediction to BBDT would be very large, and may have a negative impact on instruction cache. Therefore, despite load/store operations being less, the overall performance on real case datasets may be worse than that of the single tree variation. This was also observed in [28], an extension of [16]. For this reason, all works on prediction chose to avoid the complexity of the BBDT mask, and simplified it in various ways.

In this paper, we manage to combine the BBDT original mask and the *state prediction* paradigm by taking advantage of the code compression technique that converts a directed rooted tree into a DRAG [26]. The resulting process is modeled by a directed acyclic graph (DAG) with multiple entry points (roots), which correspond to the knowledge that can be inferred from the previous step. This guarantees a significant reduction of the machine code, which is better than that achievable by a compiler, since it can leverage the presence of equivalent actions in the trees leaves, and compress not only equal subtrees, but also equivalent ones.

Furthermore, the code which chooses the action to perform is automatically generated from the DAG with multiple entry points. The result is an incomprehensible sequence of *ifs* and *gotos*, as in the dreaded "spaghetti code". This is the reason why we christen this algorithm *Spaghetti Labeling*.

The rest of this paper is organized as follows: Section II sums up the latest contributions on CCL, Section III describes the proposed algorithm, which is then exhaustively evaluated in Section IV. Finally, in Section V conclusions are drawn.

II. Preliminaries

CCL algorithms have a clear and single result, thus algorithms differ in the number of load/store operations required to obtain it. Load operations are needed to get the input image pixel values, the provisional labels of already processed neighbours, and to access the data structures for managing the equivalences. Store operations are needed to write the provisional and final labels and to update the equivalences data structures. These often correspond to accesses in main memory or data cache, and must be considered along with the accesses required to get the instructions to be executed, which are in main memory and quickly move to the instruction cache, if its size suffices. In the following we review how these load/store operations may be reduced and how to reduce the algorithm code footprint.

A. Optimal Decision Trees

The procedure of collecting labels and solving equivalences is described in [23] as a *command execution metaphor*: the current and neighboring pixels in the mask provide a binary word, where 1 represents foreground pixels and 0 background. Each word represents a command that leads to the execution of a corresponding action, which can operate on pixels or over entire blocks with respect to the adopted mask. The possible actions are: *no action* if the current pixel/block is background, *new label* if it has no foreground neighbors, *assign* or *merge* based on the label of neighboring foreground pixels/blocks. In [29], Schutte showed that *decision tables* may conveniently be used to describe the relation between commands and corresponding actions. Additionally, Grana *et al.* [23] extended the classical concept of *AND*-decision tables to *OR*-decision tables. The former require all actions to be performed (*e.g.* perform action 3 *and* action 5 *and* . . .), while

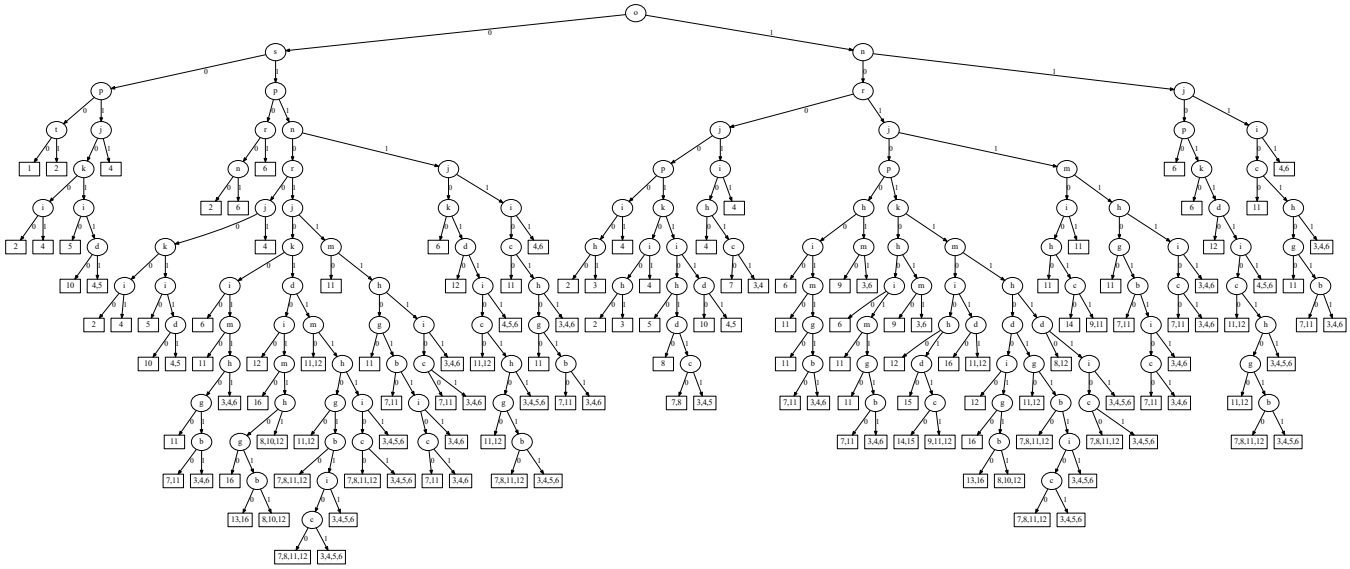


Fig. 4. Optimal DTree obtained using the algorithm presented in [24] and the mask of Fig. 1b. Best viewed online.

the latter associate to every binary word (rule) a set of possible *equivalent* actions (*e.g.* perform action 1 *or* action 7 *or* ...)

An *OR*-decision table describes a distinctive characteristic of the CCL problem: when multiple foreground neighbors in the scan mask share equivalent labels, alternative actions can be performed to label the current pixel or block. Considering the Ronsenfeld mask (Fig. 1a), the associated *AND*/*OR*-decision tables are respectively reported in Fig. 2 and Fig. 3.

The *AND*-decision table can be converted into an Optimal Decision Tree (ODT) through the use of the dynamic programming approach introduced by Schumacher *et al.* [30]. The process described by Schumacher ensures to obtain a DTree that minimizes the average number of conditions to check when choosing the correct action to be performed. This strategy has been then extended to *OR*-decision tables by Grana *et al.* in [24], as we already mentioned. In that paper, the authors prove the optimality of the conversion from *OR*-decision tables to DTree and provide a mechanism to automatically convert a DTree into running code. It may happen that, after the optimization, a leaf still contains equivalent actions, so the authors suggest that a random one can be chosen without affecting the result. This automatic procedure allows to extend the algorithm to complex masks, as the one in Fig. 1b. This mask allows the labeling of four pixels (o, p, s and t) at the same time, thus reducing the number of load/store and merge operations required. Indeed, labels equivalence is automatically solved within 2×2 blocks without requiring additional merges.

The ODT obtained with the aforementioned strategy and using Grana mask is reported in Fig. 4. Here, the total number of nodes is 136 and leaves are 137. Differently from what previously presented in [24], this tree still shows all the equivalent actions in its leaves, which will be exploited later for further optimizations.

B. State Prediction

He *et al.* [27] were the first to realize that, when the mask shifts horizontally through the image, it contains some pixels that were already inside the mask in the previous iteration.

Considering the Rosenfeld scan mask, it can be observed that pixel x will be the next s, q will be the next p, and pixel r will be the next q. See Fig. 5 as a reference. A similar observation can be drawn for the other masks. Then, in the case those pixels were indeed checked in the previous step, a repeated read can be avoided. He *et al.* [16] addressed this problem condensing the information provided by the values of already seen pixels in a configuration state, and modeled the transition with a finite state machine. The algorithm was designed for the specific task, and no provision of a general methodology is provided in the paper.

Grana *et al.* [25], instead, proposed a general paradigm to leverage already seen pixels, which combines configuration transitions with the decision trees. Algorithms that make use of a DTree usually traverse the same tree for each pixel of the input image. In that paper, authors noticed that the exploitation of values seen in the previous iteration could result in a simplification of the decision tree for the current pixel. A reduced DTree can be computed for each possible set of known pixels. Then, trees can be connected into a single graph (forest), which drives the execution of the CCL algorithm on the whole image. It is noteworthy that, for a certain position of the mask, the set of pixels that theoretically do not need

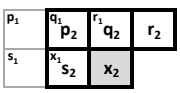


Fig. 5. Unitary horizontal shift for Rosenfeld mask during image scan. Pixels named with "1" were inside the mask in the previous iteration while pixels named with "2" are currently inside the mask: $q_1 \rightarrow p_2$, $r_1 \rightarrow q_2$, and $x_1 \rightarrow s_2$.

to be read are not only the already seen ones, but also pixels that are outside the input image, which are usually considered background. In fact, Grana *et al.* also exploited the information about the position of the mask in the image, and built special reduced DTrees that are only used in correspondence with borders. The whole optimization strategy has been applied *by hand* to the SAUF algorithms, *i.e.* using the Rosenfeld mask.

The use of reduced DTrees that leverage any possible *a priori* knowledge about pixel values has two advantages in terms of execution time. The first one is the saving of load/-store operations, which are the major performance bottleneck of this kind of CCL algorithms. The second advantage is the saving of pixel existence checks: every reduced DTree only contemplates pixels that for sure do not exceed the borders of the input image. Thus, boundary checks, that would otherwise be necessary every time a pixel is accessed to ensure it is inside the image, can be removed.

C. From Trees to DRAGs

In [19], authors noticed the existence of identical and equivalent subtrees in the optimal decision tree obtained using the algorithm by [24]. They observed that identical subtrees were merged together by the compiler optimizer, with the introduction of jumps in machine code. The result of such a merging is the conversion of a tree into a Directed Rooted Acyclic Graph, which they called DRAG.

While the code compression operated by the compiler optimizer is aimed at the reduction of code footprint, the compiler is only capable of recognizing identical pieces of code. In [19], this optimization is enhanced by merging not only identical subtrees, but also equivalent ones. The formal statement of the problem is as follows.

The set of decision trees for the set of conditions C and actions A is called $\mathcal{DT}(C, A)$. These are trees in which every node that is not a leaf has two children, also known as full binary trees. $\mathcal{N}$ is the set of nodes and $\mathcal{L}$ is the set of leaves. The condition of a node is denoted with $c(n) \in C$, with $n \in \mathcal{N}$, and the set of equivalent actions of a leaf is denoted with $a(l) \in \mathcal{P}(A) \setminus \{\varnothing\}$, with $l \in \mathcal{L}$, where $\mathcal{P}(A)$ is the power set of A. Each node n has a left subtree $\ell(n)$ and a right subtree $r(n)$, each rooted in the corresponding child of n.

Definition (Equal Decision Trees)**.** Two decision trees $t_1, t_2 \in \mathcal{DT}$, having corresponding roots r_1 and r_2, are *equal* if either:

1) $r_1, r_2 \in \mathcal{L}$ and $a(r_1) = a(r_2)$, or
2) $r_1, r_2 \in \mathcal{N}$, $c(r_1) = c(r_2)$ and $\ell(r_1)$ is *equal* to $\ell(r_2)$ and $r(r_1)$ is *equal* to $r(r_2)$.

Definition (Equivalent Decision Trees)**.** Two decision trees $t_1, t_2 \in \mathcal{DT}$, having corresponding roots r_1 and r_2, are *equivalent* if either:

1) $r_1, r_2 \in \mathcal{L}$ and $a(r_1) \cap a(r_2) \neq \varnothing$, or
2) $r_1, r_2 \in \mathcal{N}$, $c(r_1) = c(r_2)$ and $\ell(r_1)$ is *equivalent* to $\ell(r_2)$ and $r(r_1)$ is *equivalent* to $r(r_2)$.

A pair of equal or equivalent trees can be merged into a single one, to which both their parents can point. The aim of the conversion of a decision tree to a DRAG is the exploitation of such merging operations, in order to minimize the total number of nodes. One possibility consists of traversing the tree in a chosen order, and merging each subtree with every possible equal one. Since $\mathcal{DT}$ equality is a transitive relation, the result of this operation does not depend on the order chosen for traversing the tree [31].

An analogous procedure could merge equivalent trees instead, taking the intersection of actions in the corresponding leaves. However, since $\mathcal{DT}$ equivalence is not a transitive relation, this procedure would depend on the order in which the tree is traversed. Trying all possible orders would lead to the optimum, but it is clearly not feasible. Thus, the authors chose another way: first, they operate the merging of only equal trees, whose result is optimal because it does not depend on the traversing order. In this way, an initial DRAG is obtained, which keeps the entire sets of equivalent actions in the leaves. From that one, many variations can be generated by selecting a single action from leaves with more than one, in all possible ways. After the generation of all equivalent DRAG variations, the merging of equal subtrees is performed on each of them separately. Then, the one with the minimum number of nodes, which represents the optimal solution to the original problem, is selected.

III. Outline of the Proposed Method

In this paper, we propose a new CCL algorithm that combines the BBDT original mask and the state prediction paradigm, by taking advantage of the technique that merges together equal and equivalent subtrees, in order to minimize code footprint. This allows to minimize load/store and merge operations, and *if* statements required by the labeling procedure, and to increase instruction cache hits, thus improving the overall execution time.

A. State Prediction with Grana Mask

In the previous work by Grana *et al.* [25], the Rosenfeld mask is adopted, and both the forest and the graph that links trees together are manually built. In order to implement the same approach with a different and more complex mask, such as the BBDT one, we employ a generic algorithm that automatically generates forests of reduced DTrees and connects them into graphs. This approach is described in the following of this section.

1) Forest of Reduced Trees: We represent the information about an already known pixel through a *constraint*, which is an ordered pair (p, v), where p is a pixel of the mask, and v is its value. For each leaf of the complete tree, we build a set of constraints that stores the information about pixels that have been read in the path to the leaf. These constraints are modified according to the mask shift. Considering for example the simplified case with unitary shift in Fig. 5, a constraint over the pixel x will turn into a constraint on pixel s. For each leaf, a reduced tree is then generated through the application of the corresponding set of constraints to the original tree.

A reduction is performed traversing the original tree, from root to leaves, recursively. Each node that contains a condition over a pixel included in the constraints set is substituted

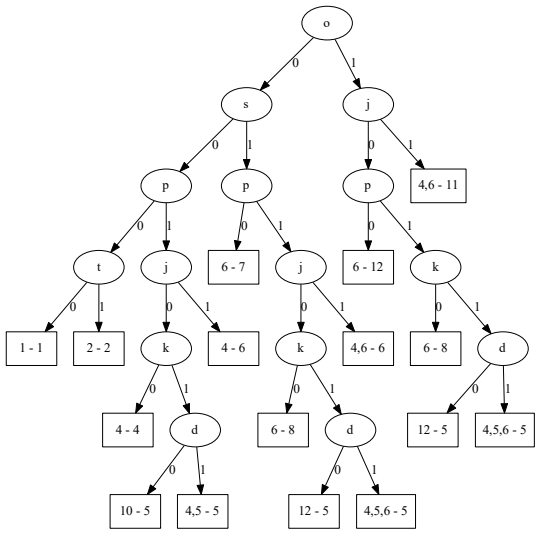


Fig. 6. Reduced tree obtained from the optimal DTree (Fig. 4) through the application of the set of constraints $\{(h,0),(n,1),(i,1)\}$. Leaves contain equivalent actions to perform (to the left) and the index of the next tree to use (to the right) separated by a dash.

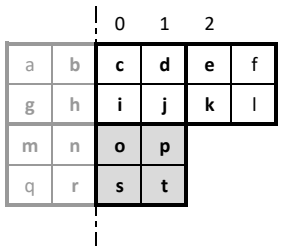


Fig. 7. Grana mask configuration at the begin of a line. Dotted line represents the left border of the image.

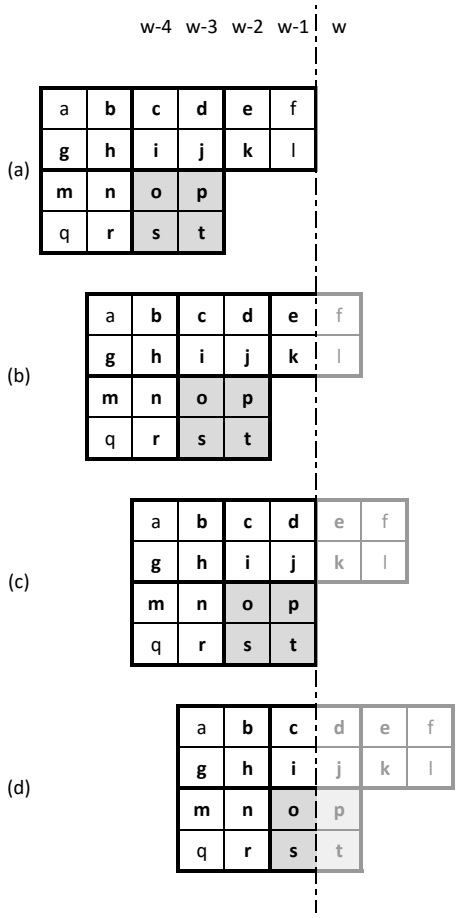


Fig. 8. Possible configurations of Grana mask when it reaches the end of a line. The dotted line represents the right border of the image. Configurations (a) and (c) will occur when image has an even number of columns. Configurations (b) and (d) are required with odd number of columns.

with its child that corresponds to the constraint value. This reduction, that is a pruning of the original tree, allows to remove the checking of already known pixels. As an example, if a constraints set contains $(i,0)$, each node of the original tree with condition i is replaced with its child associated to branch 0. Every reduced tree is given a unique index.

Obviously, after the reduction it is possible for a node of a reduced tree to have two identical branches. Those can therefore be merged together, by substituting the parent node with one of its equal children. Then, possible pairs of identical reduced trees are looked for, and duplicates are deleted. An example of a reduced tree is shown in Fig. 6.

In the CCL process, after a certain decision tree has been traversed to a leaf in order to perform an action for a certain pixel (or group of pixels), the linked tree is traversed for the next pixel of the row. We call the reduced decision trees generated in this step *main trees*, to distinguish them from ones that will be discussed further on. The set of main trees is called *main forest*.

2) Beginning of Rows: At the beginning of the row, no information is available about previously seen pixels. Thus, the complete decision tree could be used. Anyway, we know that some pixels of the mask are out of the borders of the input image. For the BBDT mask, the case is shown in Fig. 7. Since those pixels are always considered to be background, there is no need to use the complete tree. Conversely, a reduced tree to be used at the beginning of the row is built, imposing constraints that set to 0 every pixel outside the image. We call this tree *start tree*. The start tree is added to the main forest, and is considered a main tree to all effects.

3) End of Rows: An analogous reasoning can be applied to the end of rows, where the mask pixels to the right can be outside the input image. However, this case is different from the beginning of row, and presents two more issues.

The first one is that, in addition to end of row constraints, we may also have information about pixels already seen in the previous iteration. Such information has already been coded in a main tree. Therefore, for every main tree, a further reduction is performed by applying the end of row constraints, to generate the corresponding end of row tree. End of row trees from now on will simply referred to as *end trees*, and together they compose the *end forest*. During the CCL procedure, whenever the traversing of a main tree starts, a preliminary check is done on the column index, to establish whether the corresponding end tree should be used instead.

The second problem is tied to the specific mask chosen by the CCL algorithm. When using BBDT mask, we notice

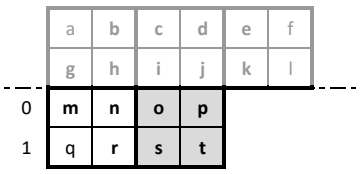


Fig. 9. Grana mask configuration at the first line. The dotted line represents the upper border of the image.

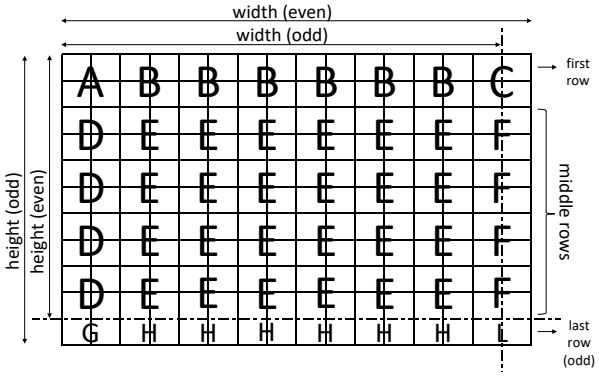


Fig. 10. This figure shows which trees/forests should be used in each part of the image. For what concerns first row, "A" is the start tree, "B" represents the main forest and "C" is the end forest which automatically handle the odd/even number of cols. "D", "E", "F" have a similar meaning but for middle rows. When image has an odd number of rows an additional group of forests is required: the end line forests. In that case "G" represents the last row start tree, "H" is the main forest and "L" is the end forest.

that end of row constraints depend on whether the number of columns of the input image is even or odd (Fig. 8). Thus, for each main tree, two different end trees are generated, one to be used when the columns are odd and one for when they are even. The *end forest* is therefore divided into two disjoint sets, that we will call *end forest even* and *end forest odd*.

The merging of identical branches and removal of duplicate trees are performed on end trees too.

4) First Row and Last Row: It is also possible to observe that when the first row is scanned, the upper part of the mask is outside of the image (Fig. 9). Similarly, if the number or rows is odd, when the last row is scanned the bottom line of the mask exceeds the lower border of the image. This observation leads to the construction of *first row forests* and *last row forests*, which can be obtained from the *main forest* and the *end forest* through the application of first line constraints or last line constraints.

Finally, in the far-fetched but possible case that the input image has only one row, a special set of forests is needed in which both first line and end line constraints are considered. Those were created and included in the algorithm, for such uncommon situations. The way in which different forests alternate during the scan of an image is depicted in Fig. 10.

B. DRAGs

Since main and end forests described in the previous section have a very large number of nodes, we exploit the merging of equal and equivalent subtrees in order to reduce code footprint and make a better use of the instruction cache. Differently

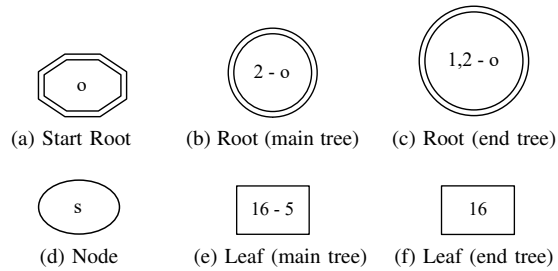


Fig. 11. (a) is the root of a start tree with the corresponding condition to check (*o* in this example). (b) and (c) are the symbols used for the roots of a generic tree inside the forest: in (b) are specified the index of the tree (2 in this example) and the condition to be checked (*o* in this example), in (c) the *group* to which the tree belongs is also reported (1 in this example). See Section III-B for details about tree *groups*. Nodes of the tree, and associated conditions to check, are represented as in (d). (c) and (f) are the symbols for leaves: the first one contains both the action to be performed (16 in this example) and the index of the next tree (5 in this example), the second one contains only the action. (b) and (c) are used inside the main forest, while (c) and (f) are required to depict end line forests.

from the original work by Bolelli *et al.* [19], however, we do not have a single tree, but three whole forests for each line case (*i.e.* first row, last row, and middle rows). In fact, equivalent subtrees can be merged together even if they belong to different trees. This approach leads to the conversion of a forest into a DAG with multiple entry points (roots). Another peculiarity of our specific case is that leaves of main trees do not only contain actions, but also a link to the root of the subsequent tree to use after the mask shift. Thus, in order for two leaves to be considered equal or equivalent, it is also required that they point to the same next tree. A consequence is that a subtree of a main tree will never have an equivalent one in the end forests, since end trees leaves do not have pointers to other trees. Anyway, it is possible for a subtree in the end forest even to have an equivalent one in the end forest odd. A choice had to be made on whether to reduce the two end forests jointly or separately. Theoretically, this choice should not impact instruction cache, because only one end forest is needed for a certain image, so pieces of code that implement trees belonging to different end forests will never conflict for a cache line during the labeling of an image. We tried both the possibilities anyway, and experimental tests did not show performance discrepancies. In the following, the two end forests will be considered separately, but every operation can be adapted to the joint option by simply substituting the two sets of end trees with a single set that represents their union.

The conversion of a forest to a DAG with multiple roots consists of two steps. The first one is the merging of equal subtrees, as described in Section II-C. The sole difference is that we do not traverse a tree only, but a set of trees one by one, and for each subtree we search for equal ones in the whole forest. This operation is guaranteed to be optimal, because it does not depend on the order in which subtrees are traversed. The second step involves equivalent subtrees reduction. Similarly as described in Section II-C, if during the tree traversal, equivalent subtrees are merged together as soon as they are found, there is the risk to change the list of equivalent actions so that now it is impossible to

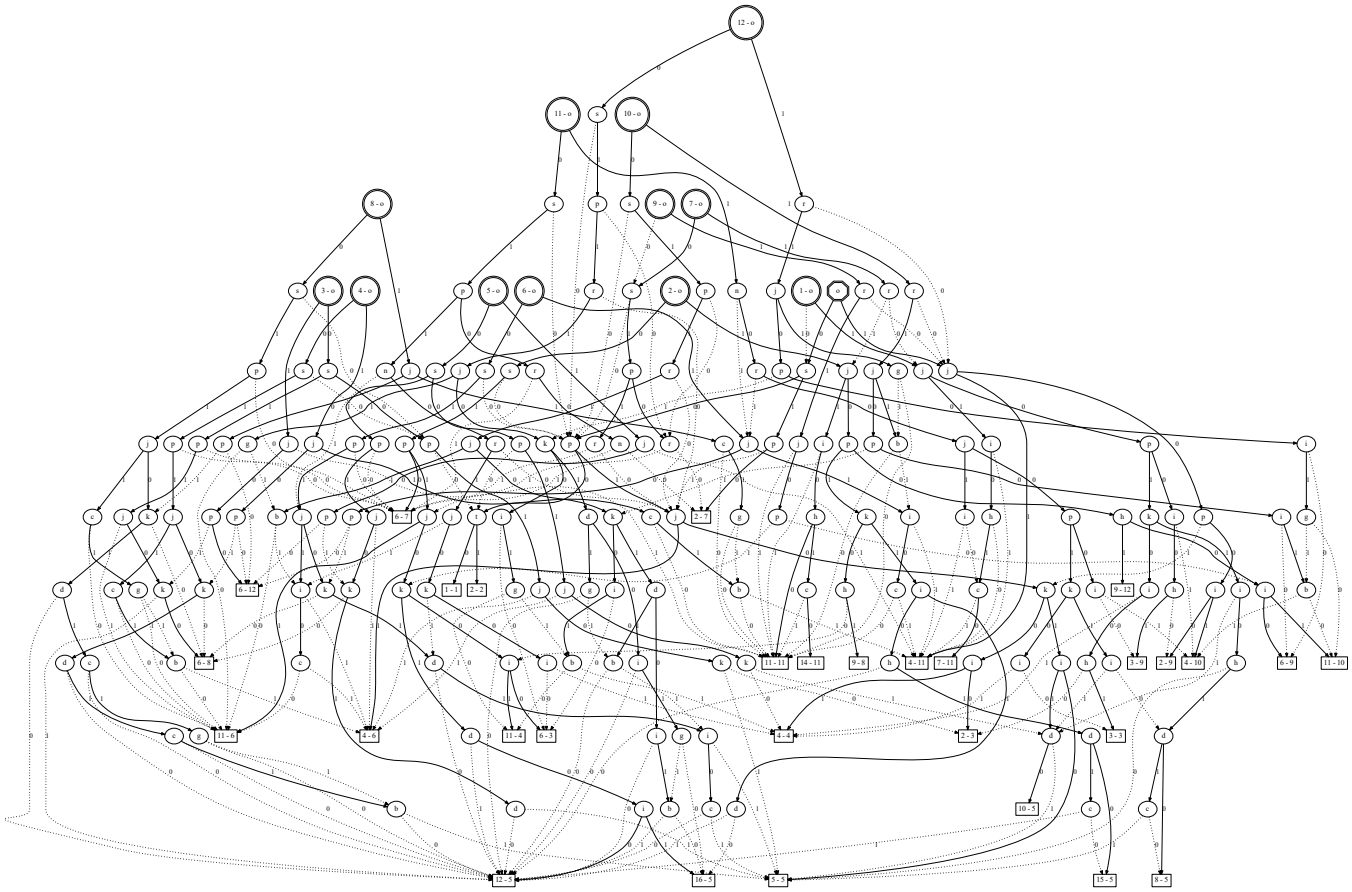


Fig. 12. Forest of main trees connected into a single DAG with multiple entry points (roots). See Fig. 11 for details about notation used. Best viewed online.

perform another substitution of larger subtrees. The exhaustive optimal solution adopted in [19] is not applicable to our case, because the amount of equivalent forests is too large for the execution time to be reasonable. To solve the problem, we use a heuristic greedy algorithm which tries to prioritize more valuable substitutions, meaning that larger equivalent subtrees get substituted before smaller ones. This is heuristic since there is no guarantee that many smaller substitutions could be performed if the large one had not been performed, but we could not spot any counter example in practice. The process follows these steps:

1) all trees in the forests are traversed and each of them is unrolled into a string with a memoizing strategy, along with a list of the possible actions and the next tree reference. Each element has also got a pointer to the subtree it represents;
2) the list of "stringized" subtrees is heuristically sorted by string length (prefer larger trees) and then lexicographically;
3) now we can move through the list and remove all the elements whose strings do not appear more than once, since no subtree shares the same conditions. This shortened list will contain possible substitutions;
4) going through the list, we now check if two entries with the same string have a non empty actions lists intersection. In this case, one of the subtrees can be replaced by a pointer to the other after intersecting the actions lists;
5) the process starts again after the removal, because now the graph structure is different and some of the smaller equivalences have already been resolved. The memoization step is not required again, since the strings are the same as before, but the list is recreated with the now reduced structure.

The process ends when no substitution is possible. The main and the end forests, specific for middle rows and converted to DAGs, are depicted respectively in Fig. 12 and Fig. 13.

C. Implementation

A tree is translated into code as a sequence of nested if and $else$ statements, that leads to the execution of exactly one action. After that, a $goto$ statement allows the execution flow to jump to the next tree.

The labeling process starts, at the beginning of each row, with the proper start tree, as shown in Fig. 10.

At the beginning of every main tree, the column index is incremented and an end of row check is performed, in order to decide whether an end tree should be used in place of the main tree. Anyway, two different end trees correspond to any main tree. This allows to cover both the case the image columns

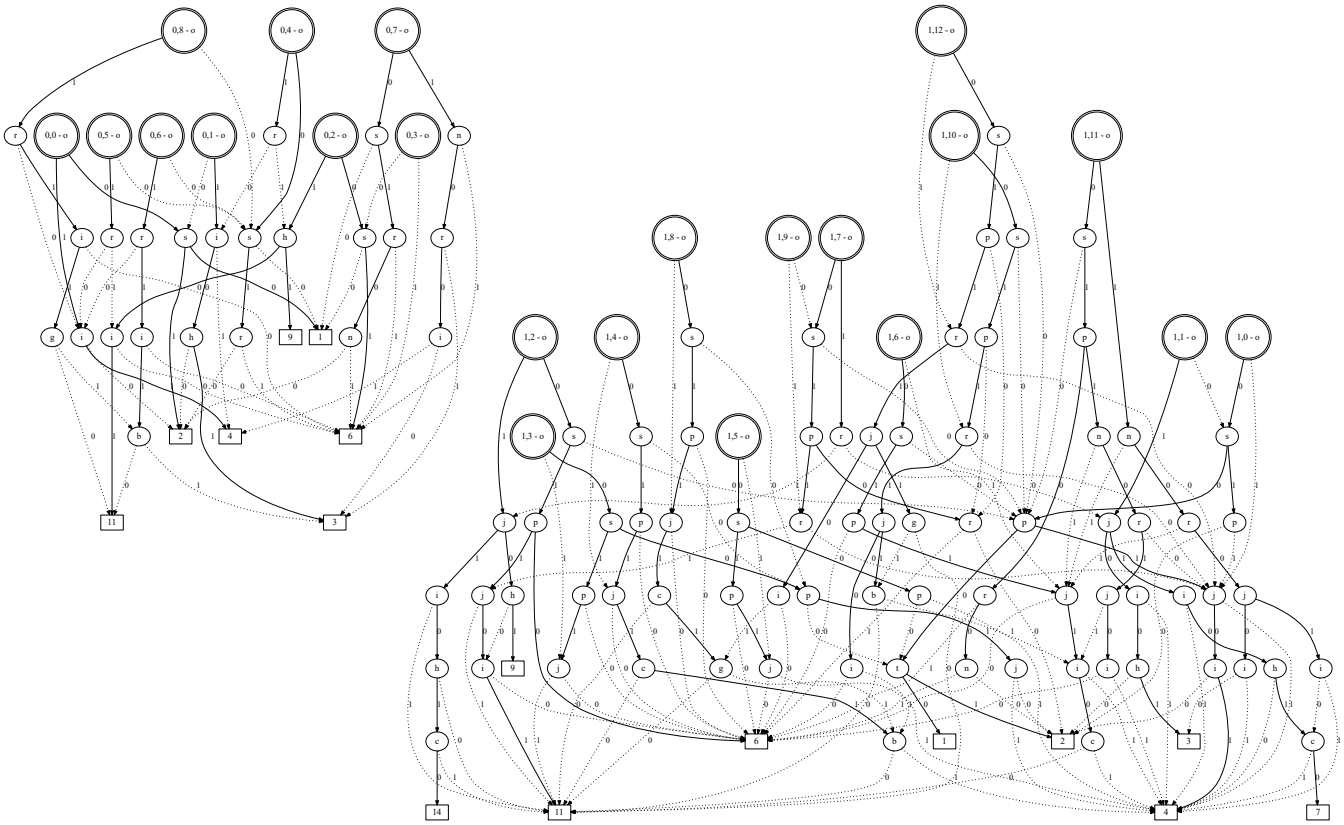


Fig. 13. Forest of end trees connected into two groups of DAGs with multiple entry points (roots). See Fig. 11 for details about notation used. Best viewed online.

are in even number and the case they are odd. Therefore, if an end of row condition is really met, a check on the number of column is needed. Then, a *goto* statement causes a jump to the proper end tree. The two possible end trees are fixed for each main tree.

After the traversal of an end tree, the column index is reset, the row index is increased and an end of column check is performed. If the image is not over yet, a *goto* statement moves the execution flow to the beginning of the appropriate start tree, and the aforementioned process continues on the next row. The *C++* like pseudo-code provided in Listing 1 exemplifies all the process.

IV. Comparative Evaluation

In this section the benefits of the proposed strategy are evaluated using YACCLAB [26], [32], [33], a widely used [34], [35] open source *C++* benchmarking framework for CCL algorithms. YACCLAB allows researchers to test state-of-the-art algorithms on real and synthetic generated datasets. The fairness of the comparison is guaranteed by compiling the algorithms with the same optimizations and by running them on the same data and over the same machine.

Experimental results discussed in the following are obtained on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz (4×32 KB L1 data cache, 4×32 KB L1 instruction cache, 4×256 KB L2 cache, and 8 MB of L3 cache) with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. All the algorithms have been compiled for x86 architecture with optimizations enabled.

The algorithms provided by YACCLAB cover most of the paradigms for CCL explored in the past. We selected the most significant ones, *i.e.* the best performing according to [26], in order to showcase the performance of the proposed algorithm. For convenience, the following acronyms will be used to identify algorithms: SAUF is the Scan Array Union Find algorithm by Wu *et al.* [36], BBDT is the Block-Based with Decision Trees algorithm by Grana *et al.* [23], CTB is the Configuration-Transition-Based algorithm by He *et al.* [16], PRED is the Optimized Pixel Prediction by Grana *et al.* [25], DRAG represents the Direct Rooted Acyclic Graph algorithm introduced by Bolelli *et al.* [19], [37]. Moreover, NULL is a lower bound limit for all CCL algorithms over a specific dataset/image, obtained by reading once the input image and writing it on the output again [26]. Finally, the proposed method is identified as *Spaghetti*.

The benchmark provides a template implementation of the algorithms over the labels solving strategy. Using different label solvers can significantly change the performance of a specific combination of dataset, algorithm and operating system. To increase the readability of charts without losing information, we select, according to [26], the labels solvers that provide the best performance on the selected environment for a specific algorithm, *i.e.* standard Union-Find (UF) for SAUF and the interleaved Rem algorithm with SPlicing

```
for (int r = 2; r < rows; r += 2) {
   int c = -2;
   // Start from root_0
root_0:
   c += 2; // Move to next block
   if (c >= cols - 2) {
      // Manage the last column/s
      goto last_column_0;
   }

   if (CONDITION_O) {
      if (CONDITION_J) {
         // Leaf: do action and
         ACTION_4
         // jump to next root
         goto root_11;
      }
      else {
...
root_6:
   c += 2; // Move to next block
   if (c >= cols - 2) {
      // Manage the last column/s
      goto last_column_6;
   }
   if (CONDITION_O) {
         // Name this node, because
         // it will be reused
         NODE_80:
         if (CONDITION_J) {
...
root_11:
   c += 2; // Move to next block
   if (c >= cols - 2) {
         // Manage the last column/s
         goto last_column_11;
   }
   if (CONDITION_O) {
         if (CONDITION_N){
               // Jump to existing subtree
               goto NODE_80;
         }
         else {
               if (CONDITION_R){
...
}
```

Listing 1. C++ like pseudo-code example of the Spaghetti algorithm. It is possible to observe the logic of block advancing (both on rows and columns), the action performed in leaves, the jump to the next tree, and the jump within conditions to reuse existing subtrees.

(RemSP) [38] for all the others.

The YACCLAB dataset [26], [39] covers most applications in which CCL may be useful, and features a significant variability in terms of resolution, image density, variance of density, and number of components. It includes six real-world datasets, and specifically:

- *3DPeS* originates from a surveillance dataset mainly designed for people re-identification [40]. Using background camera models and Otsu thresholding [41], a basic technique of motion segmentation has been applied to generate the foreground binary masks;
- *Fingerprints* dataset contains 960 fingerprints images synthetically generated or collected with low-cost optical sensors [42]. Fingerprints images have been binarized with an adaptive threshold [43] and negated to have foreground pixels with value 1;
- *Medical* is composed of 343 binary segmentations of histological images, originally published by Dong *et al.* in [44];
- *MIRflickr* [45] contains an Otsu-binarized version of the MIRflickr dataset;
- *Tobacco800* [46] is a realistic database for document image analysis research. These documents have been collected using a wide variety of equipment over time, and are characterized by a significant variability in terms of resolution;
- *XDOCS* dataset [11], [47], [48] collects a huge amount of high resolution images taken from the civil registries that are available since the constitution of the Italian state.

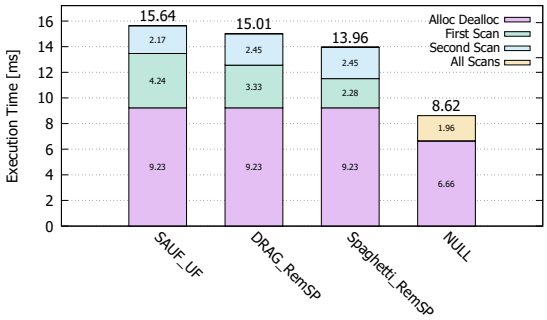


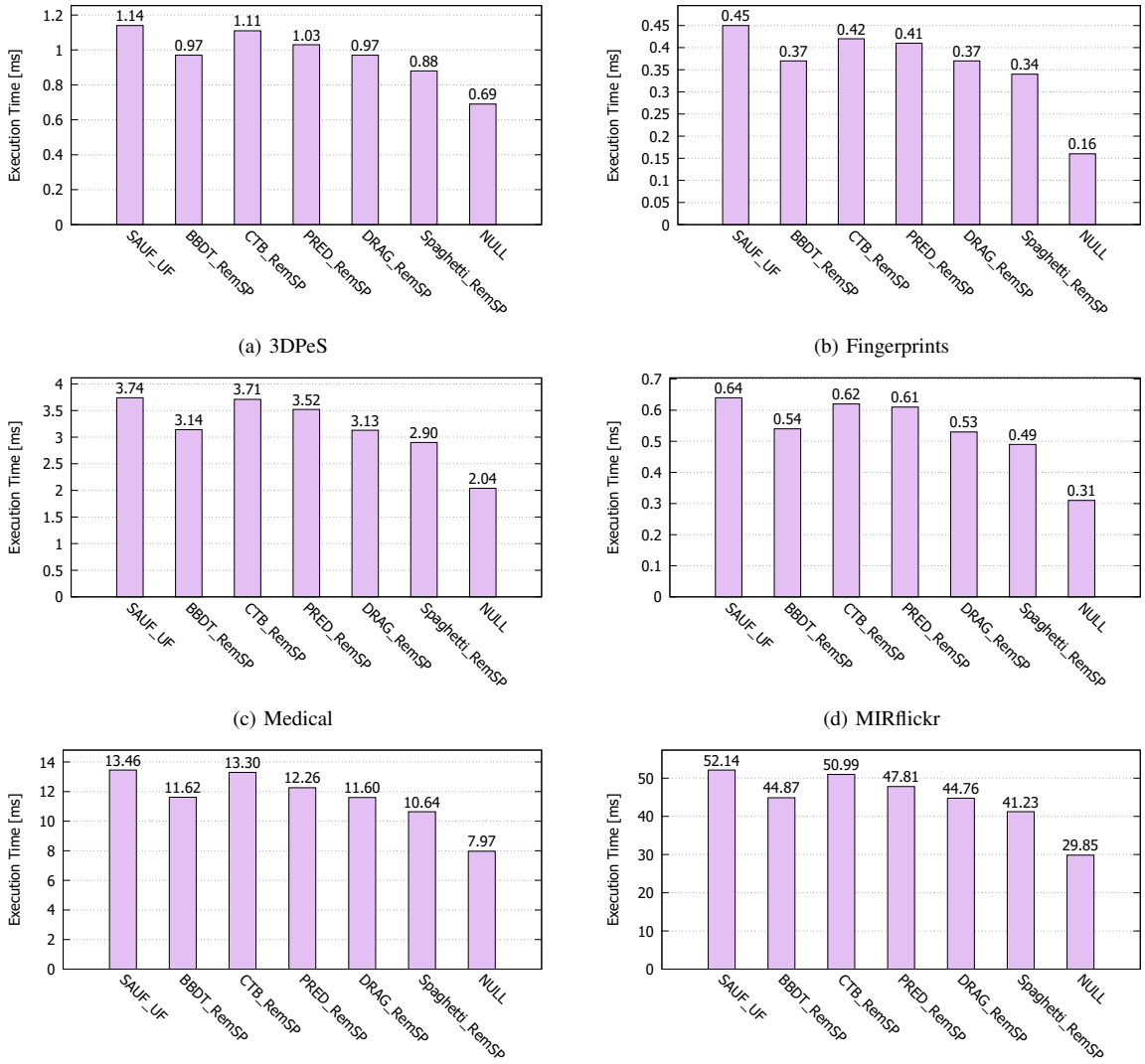
Fig. 14. Average run-time tests with steps in ms on the *Tobacco800* dataset, obtained on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. Lower is better.

A. Average Run-Time Results

For a given dataset, the set of algorithms is randomly shuffled, and each of them is run for a total of 10 iterations on each image. The minimum execution times are then considered and averaged on the dataset size [26]. Experimental results are reported in Fig. 15.

Algorithms that make use of Grana mask (BBDT, DRAG and Spaghetti) always perform better than those based on Rosenfeld mask (SAUF and PRED) or He mask (CTB), because of the lower number of merge operations and memory accesses on the output image and the equivalence data structures. Regarding the Rosenfeld mask, the advantage of PRED over SAUF is attributed to state prediction. Combining benefits of the two paradigms (block-based mask and state prediction), Spaghetti always has the lowest execution time. The improvement over DRAG, which represents the state-of-the-art of algorithms with publicly available implementations, is around 8%, which is significant considering the maturity of the problem and the small gap between state-of-the-art and the theoretical lower bound (NULL).

In order to highlight how the optimization introduced by our proposal influences the performance, a stacked bar chart obtained on the *Tobacco800* dataset is also reported in Fig. 14. In this experiment, the performance of an algorithm is evaluated splitting the allocation-deallocation (*alloc/dealloc*) time from the one required to compute CCL. Moreover, each scan involved in the labeling procedure is displayed separately. It is important to underline that *alloc/dealloc* is an upper bound of the real allocation time [26], and this explains why execution



(a) 3DPeS (b) Fingerprints

(c) Medical (d) MIRflickr

(e) Tobacco800 (f) XDOCS

Fig. 15. Average run-time tests in ms on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler (lower is better).

times in Fig.14 are higher than the ones in Fig.15e. As it can be seen, the allocation time of SAUF, DRAG and Spaghetti is the same. Indeed, they require the same data structures to compute labeling, *i.e.* the output image and the vector to solve the labels equivalences. The NULL algorithm has a faster allocation step since it does not need an additional equivalences vector. During the *first scan*, temporary labels are assigned to the output image and possible equivalences between them are stored in the equivalence vector. During the *second scan* the provisional values are replaced with final labels. For what concerns these steps the following conclusions can be draw:

- the first scan of DRAG algorithm is clearly faster than the one of SAUF and this underlines the benefit of labeling pixels 2 by 2;
- on the other hand, the second scan of DRAG (equal to the one of Spaghetti) is slower. This is linked to the fact that the algorithms based on the 2×2 mask require a bunch of *if* statements which are avoided during the first scan. Anyway, the benefits introduced in the first scan outclass the penalties in the second one;
- all the optimizations introduced with Spaghetti labeling are restricted to the first scan.

Moreover, given that allocation/deallocation time is fixed and does not depend on the algorithm, the fraction of the execution time that can be improved is a small percentage of the total execution time and this confirms again the effectiveness of the proposal.

To highlight the validity of the proposal, additional average-run time results carried out on different environments (*i.e.* different CPU architecture combined with different OS and compilers) are reported in Appendix A.

B. Load/store Operations

In Table I, the average number of load/store operations for each algorithm on the *Tobacco800* dataset is reported. The goal

TABLE I
AVERAGE NUMBER OF LOAD/STORE OPERATIONS ON THE *Tobacco800* DATASET. QUANTITIES ARE GIVEN IN MILLIONS.

Algorithm	Binary Image	Labels Image	Equivalences Vector/s	Total
SAUF_UF	4.935	14.286	4.638	23.860
BBDT_RemSP	4.942	11.586	0.120	16.648
CTB_RemSP	4.732	14.290	4.661	23.683
PRED_RemSP	4.860	14.286	4.649	23.795
DRAG_RemSP	4.942	11.586	0.120	16.648
Spaghetti_RemSP	4.902	11.586	0.120	16.608
NULL	4.604	4.604	0.000	9.208

is to observe how the choice of the mask and state prediction affects read and write operations, and thus the performance of an algorithm. It is important to notice that counts displayed in the table do not distinguish between cache hits and misses. However, they allow to draw more detailed conclusions than raw execution times.

The use of Grana mask (BBDT and DRAG) causes a small increase in the number of accesses to the input image w.r.t. the Rosenfeld mask (SAUF), but drastically reduces those to the output one and to the resolution vector/s. As expected, the code compression introduced with DRAG does not affect data accesses, but only instruction fetch, increasing instruction cache hits.

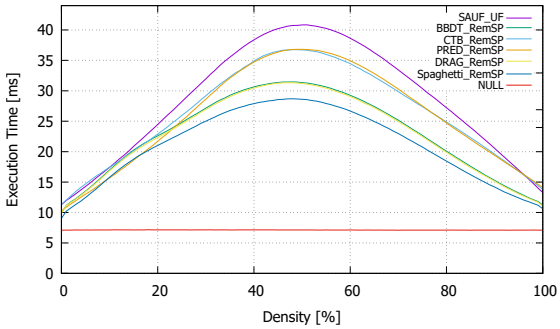
The saving of loads to the input image allowed by state prediction can be observed in the comparison between SAUF and PRED or BBDT and Spaghetti. This optimization does not affect neither the operations on the output image nor the ones on equivalence vector/s. The difference between SAUF and PRED over the equivalence vector/s is solely imputable to the different label solvers.

As of CTB, it can be observed that the use of a reduced block mask, that only labels two pixels at a time, leads to the lowest number of read operations on the input image. Nevertheless, it requires the maximum number of accesses to the output image and to the equivalence data structures.

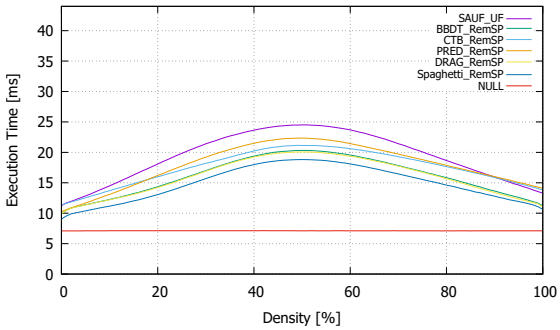
Spaghetti is overall the algorithm with the lowest number of load/store operations. This feature strictly affects performance, and explains the reason why our proposal is less time consuming than state-of-the-art alternatives.

C. Synthetic Images

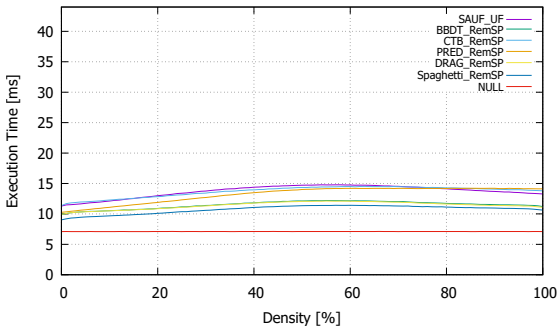
Following a common practice in literature [16], [23], [25], [26], we test the performance on images with varying density and size, taken from *density* and *granularity* datasets of the YACCLAB benchmark. *Density* is composed of black and white random noise square images with nine different foreground densities and with a resolution varying between 32×32 and 4096×4096 pixels. For each couple density-size 10 images are provided for a total of 720 samples. Images have been generated as described in [23] and have been already used to evaluate the performance of CCL algorithms in terms of scalability on the number of pixels. Experimental results obtained on this dataset are reported in Fig. 17. It can be noticed that state-of-the-art CCL algorithms are linear in the number of pixels. The difference relies on the y-intercept of the lines. The gap observable around 10^5 pixels is easily explainable taking into account that images do not fit L2 cache anymore, and require L3 cache to be employed.



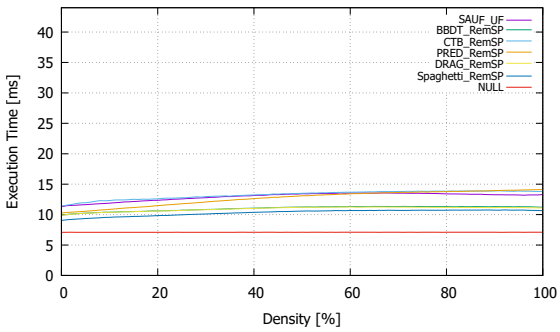
(a) $g = 1$



(b) $g = 2$



(c) $g = 8$



(d) $g = 16$

Fig. 16. Granularity results in ms on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. Lower is better.

Granularity is composed of ten different images for each value of density (from 0% to 100% with steps of 1%)

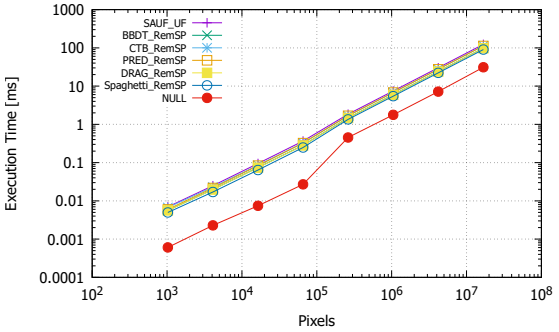


Fig. 17. Experimental results in ms on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler obtained using images of increasing size. Lower is better.

and granularity (*i.e.* dimension of the minimum foreground block). Images have a resolution of 2048×2048 and are synthetically generated with the Mersenne Twister MT19937 random number generator [49]. Results obtained with different values of granularity (1, 2, 8, and 16) are depicted in Fig. 16: the behaviour of all algorithms is strictly linked to the number of foreground pixels. More in detail, the parabolic trend can be explained considering the effects of the branch prediction unit, which heavily affects the algorithms around 50%. Focusing on *granularity*-1, it can be noticed that CTB and PRED algorithms have an analogous behaviour, since they share a very similar paradigm, though realized differently. Benefits introduced by state prediction can be appreciated when comparing these two algorithms to classic SAUF. BBDT and DRAG also have similar behaviour to each other, with the second being slightly better thanks to the code compression technique. The Spaghetti algorithm, combining all previous improvements, shows the best performance at most densities.

When the granularity grows, the execution time for middle density images decreases. This can be explained considering again the effects of the branch prediction unit: when foreground pixel blocks in the input image increase in size, the prediction of their values, which are totally random, is more accurate, thus decreasing the cost associated to failures. At any rate, considerations for *granularity*-1 are still valid at different granularities.

V. Conclusion

This paper combined all the successful techniques that improved CCL algorithms performance, producing an extremely fast solution. The proposed approach showed superior performance, beating the results obtained by all compared approaches in all settings. In order to achieve this result an automatic simplified trees generation was required, along with a solution to leverage all the savings obtainable at image edges, *i.e.* the removal of *if* statements. Moreover, a greedy algorithm was designed allowing to convert decision forests into DAGs and thus reducing the machine code size more than a compiler could ever do. This is possible thanks to the concept of equivalent subtrees. The source-code of the proposed strategy is available in [33].

References

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

[2] A. Dubois and F. Charpillet, "Tracking Mobile Objects with Several Kinects using HMMs and Component Labelling," in *Workshop Assistance and Service Robotics in a human environment, International Conference on Intelligent Robots and Systems*, 2012, pp. 7–13.

[3] R. Cucchiara, C. Grana, A. Prati, and R. Vezzani, "Computer vision techniques for PDA accessibility of in-house video surveillance," in *First ACM SIGMM international workshop on Video surveillance*. ACM, 2003, pp. 87–97.

[4] A. Abramov, T. Kulvicius, F. Wörgötter, and B. Dellen, "Real-Time Image Segmentation on a GPU," in *Facing the multicore-challenge*. Springer, 2010, pp. 131–142.

[5] A. Körbes, G. B. Vitor, R. de Alencar Lotufo, and J. V. Ferreira, "Advances on Watershed Processing on GPU Architecture," in *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*. Springer, 2011, pp. 260–271.

[6] A. Eklund, P. Dufort, M. Villani, and S. LaConte, "BROCCOLI: Software for fast fMRI analysis on many-core CPUs and GPUs," *Frontiers in neuroinformatics*, vol. 8, p. 24, 2014.

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

[8] F. Pollastri, F. Bolelli, R. Paredes, and C. Grana, "Improving Skin Lesion Segmentation with Generative Adversarial Networks," in *2018 IEEE 31st International Symposium on Computer-Based Medical Systems (CBMS)*. IEEE, 2018, pp. 442–443.

[9] ——, "Augmenting data with GANs to segment melanomaskin lesions," in *Multimedia Tools and Applications*. Springer, 2019, pp. 1–18.

[10] T. Lelore and F. Bouchara, "FAIR: A Fast Algorithm for Document Image Restoration," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 2039–2048, 2013.

[11] F. Bolelli, "Indexing of Historical Document Images: Ad Hoc Dewarping Technique for Handwritten Text," in *Italian Research Conference on Digital Libraries*. Springer, 2017, pp. 45–55.

[12] T. Berka, "The Generalized Feed-forward Loop Motif: Definition, Detection and Statistical Significance," *Procedia Computer Science*, vol. 11, pp. 75–87, 2012.

[13] M. J. Dinneen, M. Khosravani, and A. Probert, "Using OpenCL for Implementing Simple Parallel Graph Algorithms," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2011, p. 1.

[14] S. Byna, M. F. Wehner, K. J. Wu *et al.*, "Detecting atmospheric rivers in large climate datasets," in *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*. ACM, 2011, pp. 7–14.

[15] C. Grana, D. Borghesani, and R. Cucchiara, "Fast block based connected components labeling," in *2009 16th IEEE International Conference on Image Processing (ICIP)*. IEEE, 2009, pp. 4061–4064.

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

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

[18] L. He and Y. Chao, "A Very Fast Algorithm for Simultaneously Performing Connected-Component Labeling and Euler Number Computing," *IEEE Transactions on Image Processing*, vol. 24, no. 9, pp. 2725–2735, 2015.

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

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

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

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

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

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

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

[26] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, "Toward reliable experiments on the performance of Connected Components Labeling algorithms," *Journal of Real-Time Image Processing*, pp. 1–16, 2018.

[27] L. He, Y. Chao, and K. Suzuki, "An efficient first-scan method for label-equivalence-based labeling algorithms," *Pattern Recognition Letters*, vol. 31, no. 1, pp. 28–35, 2010.

[28] X. Zhao, L. He, B. Yao, and Y. Chao, "A New Connected-Component Labeling Algorithm," *IEICE Transactions on Information and Systems*, vol. E98.D, no. 11, pp. 2013–2016, 2015.

[29] L. J. Schutte, "Survey of decision tables as a problem statement technique," Computer Science Department, Purdue University, CSD-TR 80, 1973.

[30] H. Schumacher and K. C. Sevcik, "The synthetic approach to decision table conversion," *Commun. ACM*, vol. 19, no. 6, pp. 343–351, Jun. 1976.

[31] S. R. Buss, "Alogtime Algorithms for Tree Isomorphism, Comparison, and Canonization," in *Kurt Gödel Colloquium on Computational Logic and Proof Theory*. Springer, 1997, pp. 18–33.

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

[33] The YACCLAB Benchmark. Accessed on 2019-05-21. [Online]. Available: https://github.com/prittt/YACCLAB

[34] D. Ma, S. Liu, and Q. Liao, "Run-Based Connected Components Labeling Using Double-Row Scan," in *International Conference on Image and Graphics*. Springer, 2017, pp. 264–274.

[35] J. Chen, K. Nonaka, H. Sankoh, R. Watanabe, H. Sabirin, and S. Naito, "Efficient Parallel Connected Component Labeling with a Coarse-to-fine Strategy," *IEEE Access*, 2018.

[36] K. Wu, E. Otoo, and K. Suzuki, "Optimizing two-pass connected-component labeling algorithms," *Pattern Analysis and Applications*, vol. 12, no. 2, pp. 117–135, 2009.

[37] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, "Connected Components Labeling on DRAGs: Implementation and Reproducibility Notes," in *24rd International Conference on Pattern Recognition Workshops*, 2018.

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

[39] The YACCLAB Dataset. Accessed on 2019-05-21. [Online]. Available: http://imagelab.ing.unimore.it/yacclab

[40] D. Baltieri, R. Vezzani, and R. Cucchiara, "3DPeS: 3D People Dataset for Surveillance and Forensics," in *Proceedings of the 2011 joint ACM workshop on Human gesture and behavior understanding*. ACM, 2011, pp. 59–64.

[41] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[42] D. Maltoni, D. Maio, A. Jain, and S. Prabhakar, *Handbook of fingerprint recognition*. Springer Science & Business Media, 2009.

[43] J. Sauvola and M. Pietikäinen, "Adaptive document image binarization," *Pattern recognition*, vol. 33, no. 2, pp. 225–236, 2000.

[44] F. Dong, H. Irshad, E.-Y. Oh *et al.*, "Computational Pathology to Discriminate Benign from Malignant Intraductal Proliferations of the Breast," *PloS one*, vol. 9, no. 12, p. e114885, 2014.

[45] M. J. Huiskes and M. S. Lew, "The MIR Flickr Retrieval Evaluation," in *International Conference on Multimedia Information Retrieval*. New York, NY, USA: ACM, 2008, pp. 39–43.

[46] D. Lewis, G. Agam, S. Argamon, O. Frieder, D. Grossman, and J.Heard, "Building a test collection for complex document information processing," in *Proc. 29th Annual Int. ACM SIGIR Conference*, 2006, pp. 665–666.

[47] F. Bolelli, G. Borghi, and C. Grana, "Historical Handwritten Text Images Word Spotting Through Sliding Window Hog Features," in *19th International Conference on Image Analysis and Processing*, 2017.

[48] ——, "XDOCS: An Application to Index Historical Documents," in *Italian Research Conference on Digital Libraries*. Springer, 2018, pp. 151–162.

[49] M. Matsumoto and T. Nishimura, "Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

TABLE II
AVERAGE RUN-TIME TESTS IN MS ON AN INTEL(R) CORE(TM) I7-8700 CPU @ 3.20GHZ ON WINDOWS 10.0.17134 (64 BIT) OS WITH MSVC 19.15.26730 COMPILER. LOWER IS BETTER.

	3DPeS	*Fingerprints*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
SAUF_RemSP	0.876	0.394	2.889	0.519	10.267	39.784
SAUF_UF	0.876	0.394	2.890	0.522	10.264	39.772
SAUF_UFPC	1.020	0.424	3.226	0.554	11.804	44.939
SAUF_TTA	0.866	0.379	2.863	0.504	10.159	39.333
BBDT_RemSP	0.669	0.300	2.193	0.394	7.841	29.916
BBDT_UF	0.670	0.302	2.197	0.395	7.849	30.009
BBDT_UFPC	0.670	0.299	2.185	0.394	7.831	29.894
BBDT_TTA	0.685	0.299	2.224	0.393	8.007	30.471
CTB_RemSP	0.860	0.355	2.788	0.496	10.094	38.477
CTB_UF	0.861	0.357	2.793	0.500	10.104	38.522
CTB_UFPC	0.862	0.358	2.793	0.501	10.102	38.564
CTB_TTA	0.960	0.377	3.021	0.517	11.195	42.213
PRED_RemSP	0.790	0.347	2.672	0.487	9.303	36.190
PRED_UF	0.771	0.341	2.611	0.482	9.080	35.375
PRED_UFPC	0.792	0.350	2.673	0.493	9.311	36.251
PRED_TTA	0.803	0.348	2.648	0.471	9.423	36.536
DRAG_RemSP	0.670	0.299	2.169	0.390	7.815	29.785
DRAG_UF	0.687	0.303	2.224	0.396	8.028	30.539
DRAG_UFPC	0.670	0.302	2.178	0.392	7.821	29.841
DRAG_TTA	0.694	0.305	2.250	0.398	8.092	30.843
Spaghetti_RemSP	0.617	0.274	2.027	0.362	7.263	27.738
Spaghetti_UF	**0.599**	**0.271**	**1.990**	**0.360**	**7.063**	**27.051**
Spaghetti_UFPC	0.600	0.274	2.008	0.362	7.084	27.187
Spaghetti_TTA	0.603	0.273	2.030	0.363	7.120	27.385
NULL	0.451	0.111	1.353	0.205	5.140	18.614

APPENDIX A

This Appendix provides additional experimental results and allows to compare the proposed strategy with state-of-the-art algorithms from a different and exhaustive point of view. These tests have been performed on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz (6×32 KB L1 data cache, 6×32 KB L1 instruction cache, 6×256 KB L2 cache, and 12 MB of L3 cache) running Windows 10.0.17134 (64 bit) OS with both MSVC 19.15.26730 (Table II) and GCC 5.1.0 (Table III) compilers and Linux 4.18.0 (64 bit) OS with GCC 5.5.0 compiler (Table IV). This comparison highlights how the performance of an algorithm coupled with a label solver may significantly change with the environment. The RemSP label solver is not always the best choice for Spaghetti. On this specific architecture, the algorithm performs better with UF solver on Windows (both with MSVC and GCC) and with RemSP solver under Linux (with GCC).

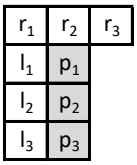


Fig. 18. Scan mask for CTBE [28]. Gray squares identify current pixels to be labeled using information extracted from white pixels.

TABLE III
AVERAGE RUN-TIME TESTS IN MS ON AN INTEL(R) CORE(TM) I7-8700 CPU @ 3.20GHZ ON WINDOWS 10.0.17134 (64 BIT) OS WITH GCC 5.1.0 COMPILER. LOWER IS BETTER.

	3DPeS	*Fingerprints*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
SAUF_RemSP	0.986	0.364	3.052	0.531	11.602	43.068
SAUF_UF	0.988	0.369	3.062	0.537	11.625	43.214
SAUF_UFPC	0.999	0.402	3.119	0.543	11.758	44.069
SAUF_TTA	1.037	0.389	3.265	0.560	12.032	45.111
BBDT_RemSP	0.735	0.342	2.338	0.422	8.523	32.575
BBDT_UF	0.731	0.330	2.275	0.404	8.477	32.232
BBDT_UFPC	0.736	0.344	2.345	0.424	8.532	32.655
BBDT_TTA	0.756	0.348	2.383	0.426	8.686	33.233
CTB_RemSP	0.697	0.319	2.492	0.473	8.524	32.987
CTB_UF	0.679	0.311	2.365	0.451	8.287	31.899
CTB_UFPC	0.673	0.321	2.466	0.481	8.271	32.245
CTB_TTA	0.692	0.322	2.512	0.476	8.426	32.846
PRED_RemSP	0.700	0.329	2.405	0.440	8.532	33.114
PRED_UF	0.693	0.313	2.352	0.428	8.404	32.388
PRED_UFPC	0.701	0.330	2.407	0.443	8.528	33.105
PRED_TTA	0.719	0.333	2.448	0.441	8.677	33.681
DRAG_RemSP	0.760	0.353	2.451	0.444	8.814	33.808
DRAG_UF	0.715	0.333	2.311	0.420	8.309	31.914
DRAG_UFPC	0.761	0.353	2.448	0.445	8.822	33.859
DRAG_TTA	0.784	0.359	2.512	0.450	9.015	34.564
Spaghetti_RemSP	0.692	0.324	2.280	0.421	8.069	31.045
Spaghetti_UF	**0.644**	**0.302**	**2.084**	**0.382**	**7.526**	**28.978**
Spaghetti_UFPC	0.693	0.327	2.296	0.425	8.083	31.210
Spaghetti_TTA	0.712	0.327	2.328	0.423	8.226	31.686
NULL	0.448	0.109	1.350	0.204	5.130	18.587

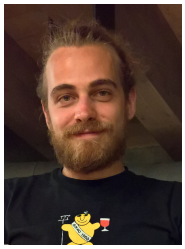
TABLE IV
AVERAGE RUN-TIME TESTS IN MS ON AN INTEL(R) CORE(TM) I7-8700 CPU @ 3.20GHZ ON LINUX 4.18.0 (64 BIT) OS WITH GCC 5.5.0 COMPILER. LOWER IS BETTER.

	3DPeS	*Fingerprints*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
SAUF_RemSP	0.922	0.350	2.845	0.491	11.010	41.181
SAUF_UF	0.926	0.365	2.959	0.502	11.156	41.890
SAUF_UFPC	0.931	0.370	2.891	0.497	11.106	41.794
SAUF_TTA	0.954	0.368	3.014	0.514	11.290	42.639
BBDT_RemSP	0.672	0.316	2.126	0.376	8.114	31.036
BBDT_UF	0.739	0.328	2.262	0.389	8.848	33.258
BBDT_UFPC	0.673	0.317	2.132	0.378	8.123	31.117
BBDT_TTA	0.727	0.330	2.223	0.390	8.651	32.753
CTB_RemSP	0.617	**0.288**	2.197	0.413	7.717	29.962
CTB_UF	0.621	0.295	2.160	0.410	7.729	30.059
CTB_UFPC	0.680	0.310	2.382	0.446	8.306	31.920
CTB_TTA	0.626	0.288	2.182	0.407	7.739	30.243
PRED_RemSP	0.633	0.293	2.172	0.388	7.816	30.507
PRED_UF	0.634	0.299	2.151	0.388	7.826	30.586
PRED_UFPC	0.635	0.297	2.170	0.393	7.828	30.592
PRED_TTA	0.643	0.306	2.182	0.393	7.885	30.940
DRAG_RemSP	0.690	0.321	2.190	0.388	8.311	31.746
DRAG_UF	0.687	0.329	2.158	0.392	8.287	31.338
DRAG_UFPC	0.690	0.323	2.196	0.389	8.304	31.808
DRAG_TTA	0.628	0.317	2.047	0.376	7.579	29.312
Spaghetti_RemSP	**0.578**	0.292	**1.910**	**0.352**	**7.013**	**27.155**
Spaghetti_UF	0.644	0.311	2.076	0.375	7.734	29.662
Spaghetti_UFPC	0.592	0.298	1.957	0.360	7.165	27.710
Spaghetti_TTA	0.651	0.314	2.084	0.375	7.762	29.826
NULL	0.400	0.099	1.190	0.173	4.801	17.595

TABLE V
AVERAGE RUN-TIME TESTS FOR OUR ALGORITHM APPLIED TO THE MASK DESCRIBED IN [28]. SETTINGS OF TABLE II.

	3DPeS	*Fingerprints*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
CTBE_RemSP	0.747	0.297	2.380	0.420	8.811	33.539
CTBE_UF	0.736	0.298	2.360	0.420	8.700	33.160
CTBE_UFPC	0.769	0.306	2.447	0.433	9.047	34.418
CTBE_TTA	0.784	0.304	2.486	0.429	9.193	34.935

Additionally, in Table V the same strategy of Spaghetti is also applied to the mask presented in [28], which is identified by CTBE (CTB Enhanced to work with three lines). Applying the proposed strategy to this mask (Fig. 18) lowers the performance, because it is unable to consider the fact that all pixels in a single block share the same label, running the mask twice for blocks which are instead a single step for the block based mask. Even if our implementation is not the original one, it is possible to see that effectively CTBE improves CTB a little, as reported in [28].



Federico Bolelli received the B.Sc. and M.Sc. degrees in Computer Engineering from Università degli Studi di Modena e Reggio Emilia, Italy. He is currently pursuing the Ph.D. degree at the AImagelab Laboratory at Dipartimento di Ingegneria "Enzo Ferrari" of Università degli studi di Modena e Reggio Emilia, Italy. His research interests include image processing, algorithms optimization, historical document analysis and medical imaging.

Stefano Allegretti received the B.Sc. and M.Sc. degrees in Computer Engineering from Università degli Studi di Modena e Reggio Emilia, Italy. He is currently a postgraduate researcher at the AImagelab Laboratory at Dipartimento di Ingegneria "Enzo Ferrari" of Università degli Studi di Modena e Reggio Emilia, Italy. His research interests include deep learning, pattern recognition, and image processing.

Lorenzo Baraldi received the Ph.D. degree cum laude in Information and Communication Technologies from Università degli studi di Modena e Reggio Emilia, Italy, in 2018. He is currently Assistant Professor at the Dipartimento di Ingegneria "Enzo Ferrari" of Università degli Studi di Modena e Reggio Emilia, Italy. He was a Research Intern at Facebook AI Research (FAIR) in 2017. He has authored or coauthored more than 30 publications in scientific journals and international conference proceedings. His research interests include image processing, video understanding, deep learning and multimedia.

Costantino Grana graduated at Università degli Studi di Modena e Reggio Emilia, Italy in 2000 and achieved the Ph.D. in Computer Science and Engineering in 2004. He is currently Associate Professor at Dipartimento di Ingegneria "Enzo Ferrari" of Università degli studi di Modena e Reggio Emilia, Italy. His research interests are mainly in computer vision and multimedia and include analysis and search of digital images of historical manuscripts and other cultural heritage resources, multimedia image and video retrieval, medical imaging, color based applications, motion analysis for tracking and surveillance. He published 5 book chapters, 34 papers on international peer-reviewed journals and more than 100 papers on international conferences.