This is the peer reviewd version of the followng article:

Connected Components Labeling on DRAGs / Bolelli, Federico; Baraldi, Lorenzo; Cancilla, Michele; Grana, Costantino. - 2018-:(2018), pp. 121-126. (24th International Conference on Pattern Recognition, ICPR 2018 Beijing, China Aug 20-24) [10.1109/ICPR.2018.8545505].

IEEE
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.

03/05/2026 05:07

(Article begins on next page)

Connected Components Labeling on DRAGs

Federico Bolelli, Lorenzo Baraldi, Michele Cancilla, Costantino Grana
Dipartimento di Ingegneria "Enzo Ferrari"
Università degli Studi di Modena e Reggio Emilia
41125 Modena, Italy
Email: {name.surname}@unimore.it

***Abstract*—In this paper we introduce a new Connected Components Labeling (CCL) algorithm which exploits a novel approach to model decision problems as Directed Acyclic Graphs with a root, which will be called Directed Rooted Acyclic Graphs (DRAGs). This structure supports the use of sets of equivalent actions, as required by CCL, and optimally leverages these equivalences to reduce the number of nodes (decision points). The advantage of this representation is that a DRAG, differently from decision trees usually exploited by the state-of-the-art algorithms, will contain only the minimum number of nodes required to reach the leaf corresponding to a set of condition values. This combines the benefits of using binary decision trees with a reduction of the machine code size. Experiments show a consistent improvement of the execution time when the model is applied to CCL.**

I. Introduction

Connected Components Labeling (CCL) is a fundamental image processing algorithm which assigns to each pixel of a connected component (object) in the input binary image a unique label, typically an integer number. Most of the research fields in computer vision, ranging from medical imaging to text analysis, exploit CCL as a preliminary operation whenever an object or its statistics need to be recognized. Differently from other tasks, CCL has an exact solution, which every algorithm should provide as output. Given the assumption that all algorithms produce the same result, the main difference among them is the execution time.

The core element of most state-of-the-art CCL algorithms is the construction of a Decision Tree (DTree) to reduce the number of pixels which need to be checked. Nevertheless, there are different data structures which could be adopted to describe the order with which the variables are checked.

In Very Large Scale Integration (VLSI) design, Binary Decision Diagrams (BDDs) have been used to model binary functions [1]. In order to have integer valued leaves, $f : \{0,1\}^n \to I$, Multi-Terminal BDDs (MTBDDs) have been introduced, where n is the number of decision variables (pixels in the mask) and I is the set of possible actions. In our case a further addition is needed, since the leaves need to have multiple alternative actions instead of just one.

In this paper we introduce a novel approach to model decision problems as Directed Acyclic Graphs with a root, which we will call Directed Rooted Acyclic Graphs (DRAGs). In this structure we can model the decision outcome as a set of equivalent actions, as it was done for the DTree of another CCL algorithm called Block Based with Decision Tree (BBDT) [2]. The advantage of this different representation is that a DRAG will contain only the minimum number of nodes required to reach the leaf corresponding to a set of condition values. In a DTree the same set of conditions may be checked in multiple subtrees, while in a DRAG, being it a graph, these could be merged together. It is clear that this does not save any condition check, but the code generated from the DRAG will include the same checks only once, sensibly reducing the number of machine instructions, thus the impact on instruction cache. In Section V we experimentally show the obtained improvement.

Fig. 1. Example of optimal DRAG for the BBDT algorithm. Rectangles identify leaves (*i.e.* actions) and ellipses identify nodes (*i.e.* conditions to check). Continuous lines represent links belonging to the original DTree while dotted lines represent links generated during the transformation process which converts the DTree into a DRAG. The full size image is reported in Fig. 7.

The rest of this paper is organized as follows: Section II sums up the latest contributions on CCL, Section III resumes the core of BBDT algorithm [2], [3] on which the method proposed in this paper is based, and Section IV deeply describes the new model. Finally, in Section VI conclusions are drawn.

II. Related Work

As for most modern computer science applications, parallel and sequential CCL algorithms have been proposed [4], but in this paper we focus our analysis on sequential algorithms only.

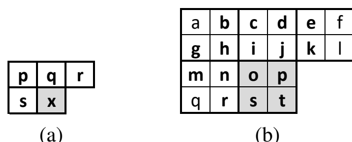


Fig. 2. (a) Rosenfeld mask for computing the label of pixel x. (b) Grana mask for computing the label of pixels o, p, s, and t.

Two classes of parallel algorithms exist for CCL: intrinsically parallel approaches and *divide and conquer* ones [5]. The latter class can also benefit of our analysis, since it uses sequential algorithms as its basic building block.

Sequential algorithms can be divided into three main categories: contour tracing (CT), multiple scan (MS) and two scans (TS) algorithms. CT algorithms [6] raster scan the input image and label the objects with the following approach: when an unlabeled boundary is found all pixels in both the contour and the adjacent background will be clockwise tagged. The object is then filled in a raster order with the same label of the contour. During the filling process, if an unlabeled boundary is found, a clockwise contour tracing is performed for internal contours. This technique proved to have linear complexity with respect to the number of labels, also because the filling of the connected components (label propagation after contour following) is cache-friendly for images stored in a raster scan order. Unfortunately, the following of contour tracing may cause a lot of cache misses, especially when connected components are big, and this limits the performance of these algorithms.

MS algorithms [7], [8] scan the input image multiple times alternatively in a forward and backward direction resolving equivalences between adjacent labels. The process ends when the last scan does not produce any change on the output image. This allows to limit the memory usage, but the number of scans and then the number of memory accesses of these algorithms can be large and negatively affect the execution time.

Usually, the TS algorithms [9] produce the best performance because they are able to generate the output labeled image with a smaller number of cache friendly memory accesses. Most of the two scan algorithms operate in three steps called *first scan*, *flatten* and *second scan*. The *first scan* goes through the input image once and assigns provisional labels to all object pixels. During this phase any possible equivalence between different labels is recorder by the *labels solver*. The *flatten* analyzes the equivalence information obtained during the previous step and determine the final labels associated to the provisional ones. Finally, the *second scan* generates the output image replacing provisional with final labels. For some tasks, the statistics on connected components are sufficient and the output labeled image is not required. In these cases the second scan can be avoided, further reducing the total execution time.

Solving equivalence between labels is a Union-Find problem [10] which can be resolved with different strategies such as classical Union-Find (UF), Union-Find with Path Compression (UFPC) [11], Three Table Array (TTA) proposed in [12] or Interleaved Rem's algorithm with splicing (RemSP) [13].

							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 associated to the Rosenfeld mask (Fig. 2a).

An exhaustive description and performance evaluation of the Union-Find strategies is reported in [14].

CCL has been studied since the dawn of Computer Vision, so what are the elements which allowed the recent efficiency improvements? Wu *et al.* noticed that only some pixels in the scanning mask (Fig. 2a) have to be checked in order to select the correct provisional label for the current pixel x [11], [15]. He *et al.* [16] modeled the problem as a Boolean algebra one and solved it with Karnaugh maps. In [2], Grana *et al.* extended the Rosenfeld mask in order to scan the input image by 2×2 blocks and reducing the number memory accesses. The problem was modeled with Decision Tables (DTab), automatically producing the associated DTree. In 2014, He *et al.* [17] introduced the Configuration Transition Based (CTB) algorithm in which they observed that during the first scan some checks can be avoided if they have been already performed in the previous step. Following this approach, Grana *et al.* in [18] proved a general paradigm to exploit already seen pixels during the scan phase, in order to minimize the number of times a pixel is accessed. In fact, the same decision tree is usually traversed for each pixel of the input image, without exploiting values seen in the previous iteration. To go beyond this limitation they compute a reduced decision tree for each possible set of known pixels: these reduced decision trees are then connected into a single graph, which rules the execution of the CCL algorithm.

III. Modelling CCL with Decision Trees

In [2] the procedure of collecting labels and solving equivalences is described by a *command execution metaphor*: the current and neighboring pixels provide a binary command word (foreground is 1 and background is 0) which leads to the execution of a corresponding action. The possible actions are: “no action” if the current pixel is background, “new label” if it has no foreground neighbors, “assign” or “merge” based on the label of neighboring foreground pixels. The relation between the commands and the corresponding actions may be conveniently described by means of a *decision table* [19]. Additionally, in [2] an extension to classical decision tables is

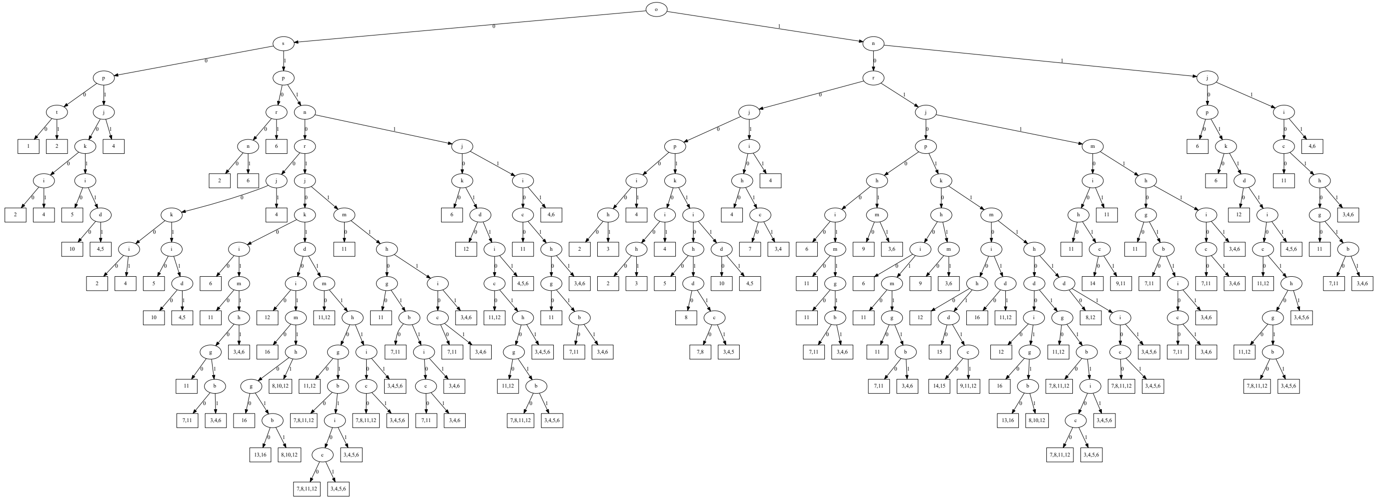


Fig. 4. Optimal decision tree obtained with the mask of Fig. 2b.

provided, *i.e.* every rule (binary word) is associated to a set of *alternative* actions. This is a peculiar characteristic of the CCL problem when multiple foreground neighbors share equivalent labels because these have already been merged in previous steps. This extension is called *OR*-decision table, opposed to the previous ones which required all actions to be performed (*e.g.* do action 1 *and* action 7 *and* ...). An example of *OR*-decision table, associated to the Ronsenfeld mask for the CCL task (Fig. 2a), is reported in Fig. 3.

An *AND*-decision table can be transformed into an optimal DTree by the use of the dynamic programming approach described in [20] by Schumacher *et al.* This process guarantees to obtain a DTree with the minimum average number of conditions to check in order to choose the correct action to be performed. Moreover, in [3] Grana *et al.* proved an optimal strategy to extend the Schumacher algorithm to *OR*-decision tables, thus allowing to convert them into DTrees and from them into running code. If any leaf still had equivalent actions, a random one could have been picked. This automatic procedure empowers the possibility to extend the algorithm to more complex masks, such as the one reported in Fig. 2b [2]. This mask has the advantage to allow the labeling of four pixels (o, p, s and t) at the same time, roughly reducing the cost of the *first scan* by a factor of four, and to reduce the number of merge operations since labels equivalence is implicitly solved within 2×2 blocks.

Fig. 4 reports the Optimal Decision Tree (ODT) obtained with the aforementioned procedure. The total number of nodes (ellipses) is 134 and leaves (rectangles) are 137. Differently from what previously presented in the literature [3], this tree still shows all the equivalent actions in its leaves. As already said, these could be chosen in any way when generating code, but we will exploit equivalences for further optimizations.

IV. From Decision Trees to DRAGs

When looking at the assembly generated by the compiler from the C source code obtained from the ODT, we observed jumps which did not correspond to *goto*s in the source code. These were generated by the compiler optimizer in order to avoid repeating pieces of code which would have been identical. In fact, by looking at the tree, it is easy to spot *identical* subtrees which do not require repetitions: the compiler is practically converting a tree into a Directed Rooted Acyclic Graph (DRAG).

The question is whether we can do something better than such a well designed tool. The answer is yes, because in order to convert the ODT into source code we removed the equivalences in the leaves, arbitrarily selecting one of the actions. Although, this substitutions may not be limited to identical subtrees: we can also compress *equivalent* subtrees.

Let us give a formal statement of the problem. We will call $\mathcal{DT}(C, A)$ the set of decision trees for the set of conditions C and actions A. These are full binary trees, *i.e.* rooted trees in which a vertex will either be a node with two children or a leaf without children. $\mathcal{N}$ 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 $l \in \mathcal{L}$. 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 first transformation from a DTree to a DRAG can be

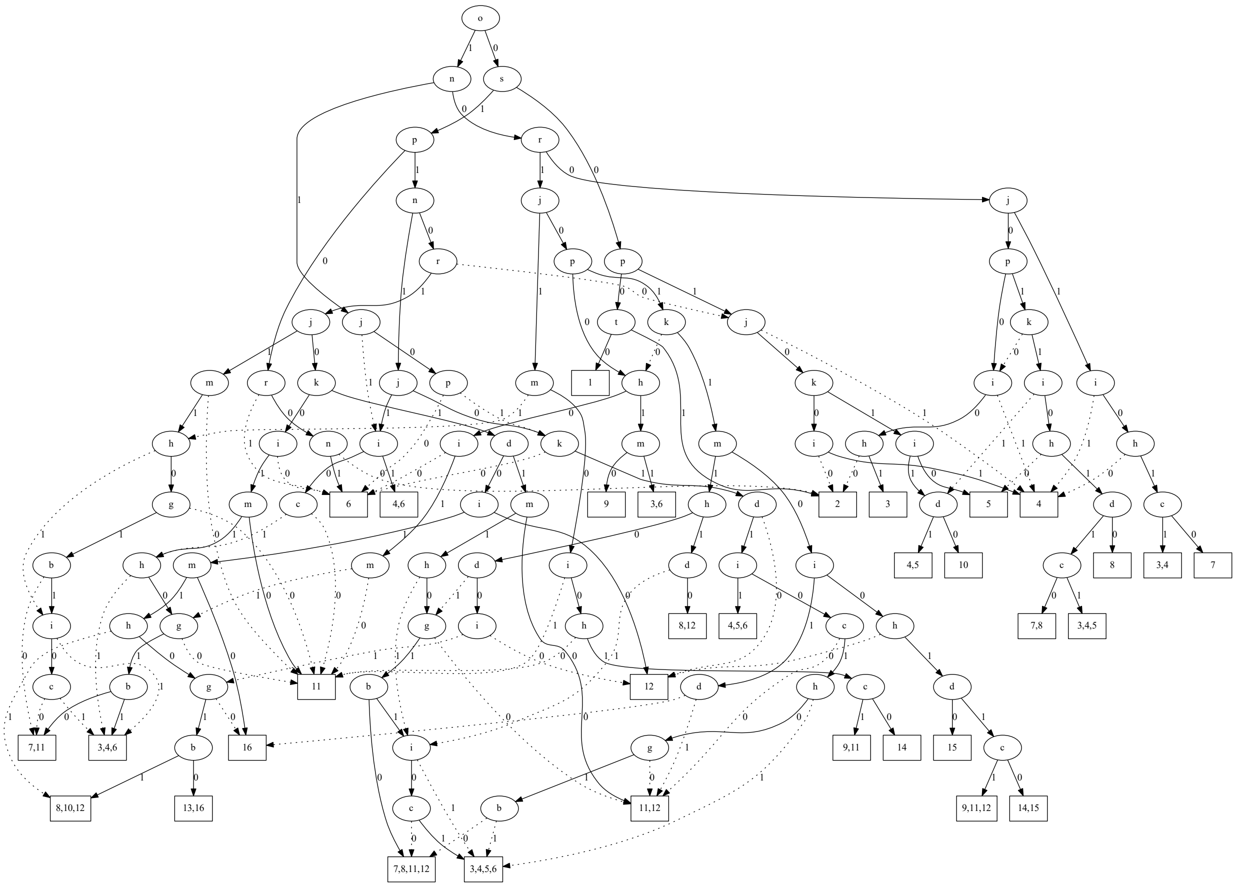


Fig. 5. DRAG obtained using the equal subtrees transformation.

performed by substituting all *equal* subtrees with a single instance, making every parent node point to that unique exemplar. Since $\mathcal{DT}$ equality is a transitive relation, we can traverse the tree and for every subtree search an equal one and immediately perform the substitution. The nice property of this transformation is that it does not depend on the order in which the original tree is traversed. The result of applying this *equal subtrees transformation* to the ODT of Fig. 4 is shown in Fig. 5. This DRAG has 86 nodes and can be automatically converted to code by generating the subtree code only for "continuous" arcs and using *goto*s for "dotted" ones.

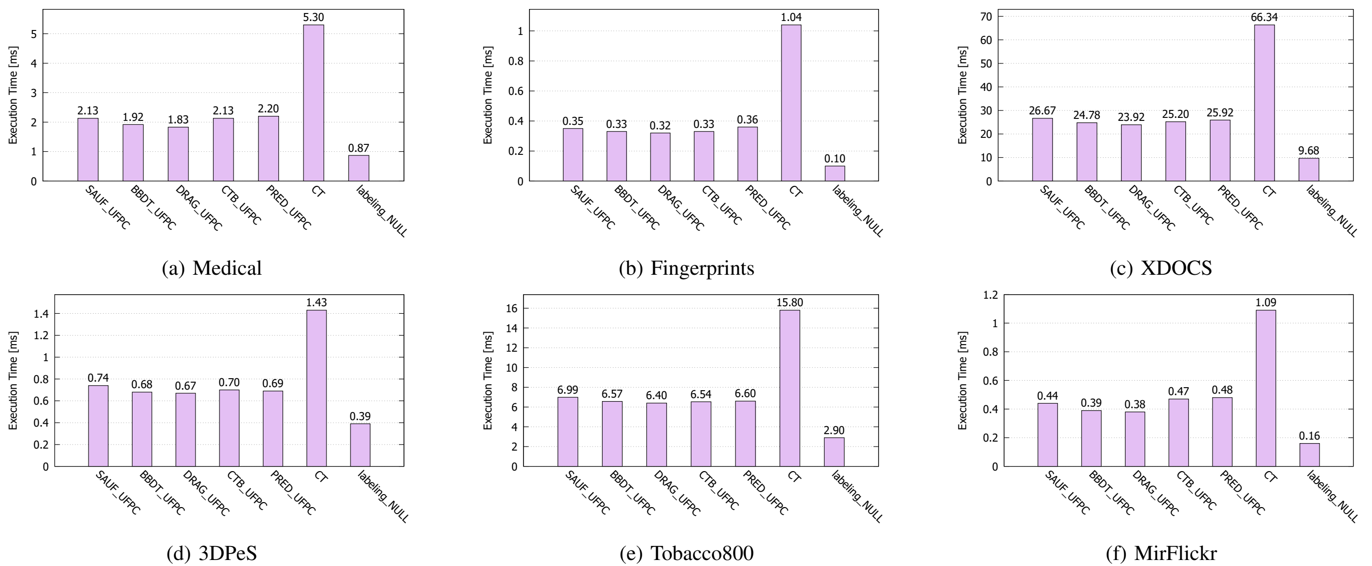
An even better transformation (*i.e.* with less nodes) is obtained by performing the same procedure, substituting equality with equivalence, and taking the intersection of actions in the corresponding leaves. Since all equal subtrees are also equivalent, all previous substitutions will be performed, plus, possibly, some more. Unfortunately, $\mathcal{DT}$ equivalence is not transitive, and it therefore depends on the order of traversal. Taking an intersection earlier may hinder the possibility of doing a better choice later, thus of minimizing the number of nodes. Of course, trying all possible traversal orders is unfeasible.

We already noticed that equal subtrees need to be substituted also when using the equivalence transformation, so instead of starting from the original tree, it is reasonable to start from the DRAG obtained by applying the equal subtrees transformation. From this DRAG we can obtain many variations by selecting a single action from leaves with more than one, in all possible ways. In our particular case (Fig. 5), 11 leaves have two choices, 5 have three and only 2 have four. This gives a total of 7 962 624 different DRAGs. Since the number is not absurd, it is possible to reduce all the DRAGs with the equal subtrees transformation, and keep the one with the minimum number of nodes. This is also an optimal solution of the original problem. The result of applying such a reduction to the DRAG of Fig. 5 is shown in Fig. 7. The final DRAG has 72 nodes and 15 leaves with no intersection.

As already said, getting to a leaf still requires all the original checks, so the benefit of implementing decisions with DRAGs is that of reducing the code footprint. The original tree required $2634B$ while the optimal DRAG version only $1919B$. The effects of such saving are reported in the following section.

V. Experimental Results

In order to evaluate the benefit of the proposed strategy we tested the DRAG algorithm using YACCLAB, an open source C++ benchmarking framework for CCL algorithms. The original version of the benchmark has been presented in [21], and it allows to test state-of-the-art algorithms on a



(a) Medical (b) Fingerprints (c) XDOCS

(d) 3DPeS (e) Tobacco800 (f) MirFlickr

Fig. 6. Experimental results obtained on an Intel Core i7-4770 CPU @ 3.40GHz, running Linux with GCC 7.2.0 with the YACCLAB benchmark. For each dataset and algorithm the average execution time in ms is reported (lower is better).

wide range of datasets covering most of the fields in which CCL could be exploited. 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. The current version of the benchmark[1] provides a template implementation of the algorithms over the labels solving strategy, but those for which the labels solver is built-in. Tests revealed that the impact of the labels solver on the overall performance is strictly limited for fastest algorithms, for this reason we will only report results obtained with the UFPC solver [11].

In the following, we will use acronyms to refer to the compared algorithms: CT is the Contour Tracing approach by Fu Chang *et al.* [6], SAUF is the Scan Array Union Find algorithm by Wu *et al.* [15], BBDT is the Block Based with Decision Trees algorithm by Grana *et al.* [2], CTB is the Configuration-Transition-Based algorithm by He *et al.* [17], and PRED is the Optimized Pixel Prediction by Grana *et al.* [18]. Moreover, labeling_NULL is a lower bound limit for all CCL algorithms, obtained by reading once the input image and writing it on the output again [22].

Fig. 6 compares the average execution time of the aforementioned algorithms on six different datasets [22]: a collection of histological images with an average amount of 1.21 million pixels to analyze and 484 components to label (Medical), fingerprint images collected by using low-cost optical sensors or synthetically generated with an average of 809 components to label (Fingerprints), high resolution historical document images with more than 15000 components and a low foreground density (XDOCS), a dataset for people detection, tracking, action analysis and trajectory analysis with very low foreground density and few components to identify (3DPeS), a selection of documents collected and scanned using a wide variety of equipment over time with a resolution varying from 150 to 300 DPI (Tobacco800), and a large set of standard resolution natural images taken from Flickr (MirFlickr). The test was run on an Intel Core i7-4770 CPU @ 3.40GHz (4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache) with Linux OS and GCC 7.2.0 compiler enabling O3 flag. The behavior is practically equal on all datasets, with DRAG being always the winner. The second best is nearly always the version using BBDT, with the ODT previously shown. The lower impact on the instruction cache is beneficial, even in this case, in which the amount of available memory is much larger than required.

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

VI. Conclusion

We have shown a strategy to model the processing of binary image patterns as a Directed Rooted Acyclic Graph. This has all the benefits of using Decision Trees, while reducing the machine code size more than a compiler could ever do, since it would miss part of the information.

The missing step for this work is the ability of including the prediction stage introduced in [17] within a unique framework, and being able of solving the complete problem in feasible time. Another extension is the application of DRAGs to other binary image problems such as thinning or mathematical morphology [23].

References

[1] S. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*. Springer Science & Business Media, 1996.

[2] 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.

[3] 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.

Fig. 7. Example of optimal DRAG for the BBDT algorithm.

[4] H. M. Alnuweiri and V. K. Prasanna, "Parallel Architectures and Algorithms for Image Component Labeling," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 14, no. 10, pp. 1014–1034, Oct. 1992.

[5] F. Bolelli, M. Cancilla, and C. Grana, "Two More Strategies to Speed Up Connected Components Labeling Algorithms," in *International Conference on Image Analysis and Processing*. Springer, 2017, pp. 48–58.

[6] F. Chang, C.-J. Chen, and C.-J. Lu, "A linear-time component-labeling algorithm using contour tracing technique," *Computer Vision and Image Understanding*, vol. 93, no. 2, pp. 206–220, 2004.

[7] R. Haralick, "Some neighborhood operators," in *Real-Time Parallel Computing*. Springer, 1981, pp. 11–35.

[8] K. Suzuki, I. Horiba, and N. Sugie, "Linear-time connected-component labeling based on sequential local operations," *Computer Vision and Image Understanding*, vol. 89, no. 1, pp. 1–23, 2003.

[9] A. Rosenfeld and A. Kak, *Digital picture processing*, ser. Computer science and applied mathematics. Academic Press, 1982, no. v. 1.

[10] M. B. Dillencourt, H. Samet, and M. Tamminen, "A general approach to connected-component labeling for arbitrary image representations," *Journal of the ACM (JACM)*, vol. 39, no. 2, pp. 253–280, 1992.

[11] 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.

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

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

[14] M. Patwary, J. Blair, and F. Manne, "Experiments on union-find algorithms for the disjoint-set data structure," *Experimental Algorithms*, pp. 411–423, 2010.

[15] 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.

[16] L. He, Y. Chao, K. Suzuki, and K. Wu, "Fast connected-component labeling," *Pattern Recognition*, vol. 42, no. 9, pp. 1977–1987, 2009.

[17] 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.

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

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

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

[21] 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.

[22] 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.

[23] C. Grana, D. Borghesani, and R. Cucchiara, "Decision Trees for Fast Thinning Algorithms," in *20th International Conference on Pattern Recognition*, Aug. 2010, pp. 2836–2839.