One DAG to Rule Them All

Federico Bolelli, *Member, IEEE,* Stefano Allegretti, and Costantino Grana, *Member, IEEE,*

Abstract—In this paper, we present novel strategies for optimizing the performance of many binary image processing algorithms. These strategies are collected in an open-source framework, GRAPHGEN, that is able to automatically generate optimized C++ source code implementing the desired optimizations. Simply starting from a set of rules, the algorithms introduced with the GRAPHGEN framework can generate decision trees with minimum average path-length, possibly considering image pattern frequencies, apply state prediction and code compression by the use of Directed Rooted Acyclic Graphs (DRAGs). Moreover, the proposed algorithmic solutions allow to combine different optimization techniques and significantly improve performance. Our proposal is showcased on three classical and widely employed algorithms (namely Connected Components Labeling, Thinning, and Contour Tracing). When compared to existing approaches —in 2D and 3D—, implementations using the generated optimal DRAGs perform significantly better than previous state-of-the-art algorithms, both on CPU and GPU.

Index Terms—GRAPHGEN, Directed Rooted Acyclic Graphs, Optimal Decision Trees, Decision Tables, Connected Components Labeling, Thinning, Chain-Code.

1 Introduction

Deep Learning, and (Convolutional) Neural Networks (CNN) in general, whose growth in popularity began in the early 2010s, have marked a shift of Computer Vision, permeating most of the academic research fields of the last decade. Thanks to their ability of learning a hierarchical representation of raw input data without relying on handcrafted features, CNNs have rapidly become a methodology of choice for analyzing medical images [1], [2], [3], [4], [5], perceiving and elaborating an interpretation of dynamic scenes [6], [7], [8], [9], [10], handwriting analysis and speech recognition [11], [12], surveillance, traffic monitoring and autonomous driving [13], [14], [15], [16], people tracking [17], [18], skeletonization [19], image synthesis [20] and so on. This became possible thanks to the increase of processing capabilities, aided by the fast development of Graphics Processing Units (GPUs), and thanks to the collection of massive amounts of datasets [16], [17], [21], [22], required during the training of the models.

Nowadays, numerous start-ups and new industrial applications come to life thanks to deep learning. Therefore, important questions come to mind: "Is computer vision and binary image processing without machine learning still worth it?", "How significant is the improvement of these kind of algorithms, both in terms of performance and accuracy, in the *deep learning era*?". As a matter of fact, most of the state-of-the-art solutions on the aforementioned research fields exploit binary image processing algorithms as fundamental pre- or post-processing steps to get to the final results [5], [13], [15], [17], [23] or to prepare training data [20]. When segmenting images, a highly relevant task in medical imaging [2], *Connected Components Labeling* (CCL) is usually exploited together with voting strategies [24] to remove noise and produce the final segmentation map [5] or to count objects [13]. Thinning, instead, is often used together with contour-tracing, morphological operators, and CCL, whenever a compact representation of the objects inside an image is required [20], as in fingerprint analysis [25], vasculature geometry detection [26], [27], and road mapping [15].

Therefore, also deep learning pipelines can benefit from efficient implementations of binary image processing algorithms. Moreover, given that image processing algorithms represent the base step of many real-time applications [28], [29], they are required to be as fast as possible. Thus, research in the field moved towards optimizing the performance of these algorithms, i.e., the execution speed.

In this paper, we introduce a novel suite of algorithms that allows to automatically apply the most appropriate optimization strategies to any problem modelled with Decision Tables (DTs), with further techniques applicable specifically to binary image processing problems. These algorithms are released with this paper as an open-source framework, called GRAPHGEN (the all encompassing GRAPH GENerator), which takes a definition of the problem, in terms of conditions that need to be checked and actions to be performed, and produces state-of-the-art solutions, directly providing the optimized C++ source code.

To showcase the capabilities of our proposal, we selected the three aforementioned fundamental image processing algorithms: connected components labeling, image skeletonization (*thinning*) and contours extraction, also known as *chain-code* extraction. We thus demonstrate the ability of the framework to automatically generate the previous state-of-the-art algorithms, starting only from the formal problem definition, and then further enhance their performance, thus setting the new reference. To prove the generality of the proposed techniques, the presented benchmarks are not limited to 2D, but also include 3D image processing algorithms and GPU applications. Moreover, the possibility of applying GRAPHGEN to additional binary image processing algorithms, such as morphological operators, is described in Appendix A, available online.

• *The authors are with the Dipartimento di Ingegneria "Enzo Ferrari," Università degli Studi di Modena e Reggio Emilia, Italy. E-mail: {name.surname}@unimore.it*

Our contributions can be summarized as follows:

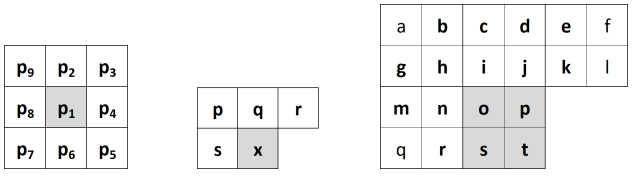
1) A novel suite of algorithms that allows to generate extremely optimized code for any problem that can be formalized with decision tables.
2) GRAPHGEN, a generalized open-source framework that integrates all the proposed algorithmic solutions, and allows to automatically generate optimized code for the selected task.
3) An improvement to the compression strategy of trees and forests described in [30].
4) New GRAPHGEN-generated binary image processing algorithms, which significantly improve state-of-the-art on 2D and 3D CCL, Thinning and Chain-Code, both in CPU and GPU environments.
5) An extensive set of experiments highlighting strengths and weaknesses of the GRAPHGEN-generated algorithms on different scenarios.

The rest of the paper is organized as follows: in Section 2 the framework is presented, with an explanation of the rationale behind the optimization strategies involved. Three different applications of GRAPHGEN are presented in Section 3, together with a review of the advances of the last decades on the fields. Finally, in Section 4 conclusions are drawn. The source code of the GRAPHGEN framework is available in [31].

2 One DAG to Rule Them All: GRAPHGEN

In the analysis of binary images, most algorithms share the same general structure: for each (group of) pixel x, some *action* must be performed, which depends on the value of x and its neighborhood. The term *action* is a general concept to describe specific algorithm operations: for Connected Components Labeling, it could be the recording of the equivalence between two label classes and the assignment of a provisional label to x; for thinning, it could be the removal of x, and so on. The set of pixels whose value can condition the action constitutes the *mask* (Fig. 1). Different algorithms will obviously perform diverse actions based on the neighborhood, but even when the task is the same, algorithmic solutions may differ in the neighborhood exploration.

Given the simplicity of the operations to be performed, one of the main elements to keep in mind is the number and the order of data load/store operations, which affect performance the most. Therefore, cache and branch prediction are critical elements that have to be considered in assessing the computational requirements of these algorithms.



(a) 3x3 (b) Rosenfeld (c) Grana

Fig. 1. Example of scan masks. Gray squares identify current pixels to be labeled using information extracted from white pixels. (a) is the classical mask adopted when exploring a 3×3 neighborhood in 8-connectivity, (b) and (c) are improved versions commonly employed in CCL.

Many implementations available in literature have already observed this, providing solutions to perform cache friendly accesses, limiting the number of conditional jumps or reusing the information of already read pixels when the mask moves through the image. Most of them implement ad hoc solutions, specifically designed for a given task, without providing general strategies. GRAPHGEN introduces a unified approach, able to automatically apply and combine the most effective solutions to access, at each step of an algorithm, only the pixels which are strictly required, while reducing the corresponding machine code.

2.1 Modeling Algorithms with Decision Tables

The class of algorithms GRAPHGEN deals with can be described with a *command execution metaphor* [32]. Values of pixels in the mask constitute a rule (binary string), which is associated to an action or, in general, a set of equivalent actions.

Considering a mask with L pixels, the set of possible rules is a L-dimensional boolean space denoted by R, where each element r has a probability p_r to occur, with $\sum_{r \in R} p_r = 1$. Given a set of actions A, the linking between rules and actions is represented by a function $\mathcal{DT} : R \rightarrow \mathcal{P}(A) \setminus \{\emptyset\}$, that can be described with an *OR*-decision table.

Given the decision table, an algorithm can simply check the value of every pixel in the mask, identify the rule, and find the action to perform in the corresponding column (Fig. 2). The *OR*-decision table can be easily translated into a Look-Up Table (LUT) where each rule is an index mapping to a vector of equivalent actions.

If the same action is associated to multiple rules, it may not be necessary to know all bits of the rule to identify the correct action. As an example, if we consider a mask with 3 pixels, p, x, q, and both rules 110 and 111 lead to action a, when $p = 1$ and $x = 1$ the action to be performed is already known, and there is no need to check q. Consequently, some processing time can be saved by removing unnecessary pixel checks, and exploiting a strategy that stops accessing pixels of the mask after it has gathered enough information to identify the correct action. A directed binary rooted tree, where each node is a condition (pixel), and each leaf contains an action, is an example of such a strategy. We call it Decision Tree, or DTree. The problem of building decision trees from a binary decision table has been addressed by Schumacher and Sevcik in [33] with a

Conditions	x	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	p	-	0	1	0	0	0	1	1	1	0	0	0	1	1	1	0	1
	q	-	0	0	1	0	0	1	0	0	1	1	0	1	1	0	1	1
	r	-	0	0	0	1	0	0	1	0	1	0	1	1	0	1	1	1
	s	-	0	0	0	0	1	0	0	1	0	1	1	0	1	1	1	1
Actions	no op.	1																
	new		1															
	p			1				1		1				1	1			1
	q				1			1			1	1		1	1		1	1
	r					1					1			1			1	1
	s						1			1		1			1		1	1
	p + r								1							1		
	r + s												1			1		

condition outcomes

action entries

Fig. 2. *OR*-decision table for the Rosenfeld mask adopted in CCL.

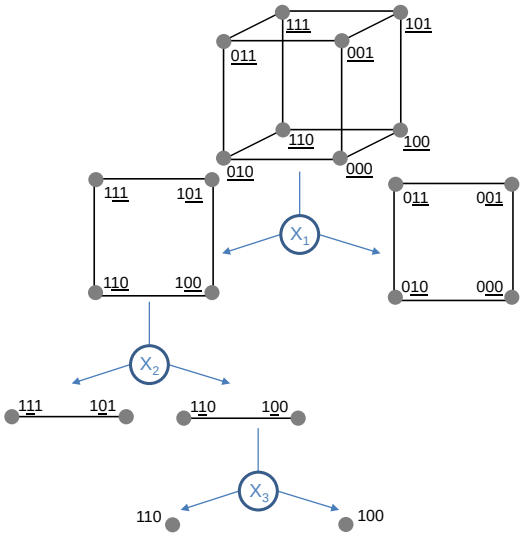


Fig. 3. Example of 3-dimensional hypercube partitioning. In this case splits are performed, in order, on index $j = 0$, $j = 2$, and $j = 1$ of the k-cubes. Underlined values represent the concept of indifference.

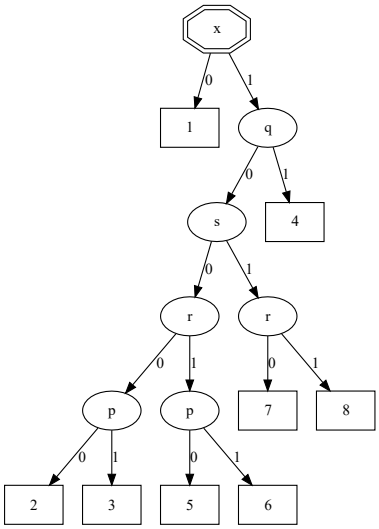


Fig. 4. Optimal DTree obtained using the algorithm presented in Section 2.1 and starting from the *OR*-decision table of Fig. 2. Internal nodes (ellipsis) represent the conditions to be checked, and leaves (rectangles) contain the actions to be performed, which are identified by integer numbers. The root of the tree, also a condition, is represented by a octagon.

dynamic programming approach, and extended in [34] to *OR*-decision tables.

The conversion of a decision table (with L conditions) to a DTree can be interpreted as the partitioning of an L-dimensional hypercube (L-cube in short), where vertexes correspond to the 2^L rules in R. We define a set $K \subseteq R$ as a k-cube if it is a cube in $\{0,1\}^L$ of dimension k. The k-cube can be represented as a L-vector containing k dashes (-) and $L-k$ values 0 and 1, where dashes represent the concept of indifference. The positions of dashes identify a set called D_K, while $P_K = \sum_{r \in K} p_r$ is the occurrence probability of the cube. Given a decision table $\mathcal{DT}$, set A_K contains the common actions to all rules in K: $A_K = \cap_{r \in K} \mathcal{DT}(r)$.

Definition 1 (Decision Tree). Given a $\mathcal{DT}$ and a k-cube K, a Decision Tree for K is a binary tree T where:

1) Each leaf ℓ corresponds to a k-cube, denoted by $K_\ell \subseteq K$, and the set of K_ℓ is a partition of K;
2) Each leaf ℓ has a non empty set of actions A_{K_ℓ}, associated to cube K_ℓ by $\mathcal{DT}$;
3) Each internal node is labeled with an index $i \in D_K$ and is weighted by w_i (which represents the cost of testing the i-th condition), with left and right outgoing edges labeled respectively with 0 and 1;
4) Root to leaf paths uniquely identify cubes associated to leaves by means of nodes and edges labels.

A tree for a k-cube K can be recursively built in this way: select an index $j \in D_K$ and label the root of the tree with j, divide cube K into two cubes $K_{j,0}$ and $K_{j,1}$, with dash in position j respectively set to 0 and 1, and recursively build the two subtrees from $K_{j,0}$ and $K_{j,1}$, stopping when a cube has a non-empty associated set of actions (i.e., $A_K \neq \emptyset$). A simple example of a 3-cube partitioning is reported in Fig. 3.

Multiple decision trees can be constructed from the same k-cube, and each can be evaluated on the basis of the average amount of condition tests that it allows to save with respect to the LUT, which represents the *gain* of the tree. This gain is defined by the following formula:

$$gain(T) = \sum_{\ell \in \mathcal{L}} \left(P_{K_\ell} \sum_{i \in D_{K_\ell}} w_i \right) \quad (1)$$

A tree with maximum gain for a k-cube is called Optimal Decision Tree (ODTree), and can be built by means of a recursive procedure. At step 0, all 0-cubes are associated to a gain of 0. At step n, the algorithm builds all possible n-cubes, each of them obtainable by merging two $(n-1)$-cubes in n different ways, and keeps track of the maximum gain obtainable for every n-cube S:

$$Gain_S = \max_{i \in D_S} (Gain_{S_{i,0}} + Gain_{S_{i,1}} + \delta w_i P_S) \quad (2)$$

Where δ equals 0 if $A_S = \emptyset$ and 1 otherwise. The procedure stops when $n = k$, and the optimal decision tree is constructed by tracing back through the merges that produce the maximum gain at each n-cube, until a cube S with $A_S \neq \emptyset$ is found, which is a leaf. Fig. 4 depicts the ODTree generated by means of the described approach starting from the *OR*-decision table of Fig. 2. Grana *et al.* proved the correctness of the algorithm in [34].

In order to provide an implementation, we need to assign the occurrence probability p_r for all rules. The simplest approach consists of considering every rule to be equally probable (i.e., $p_r = 2^{-L}, \forall r \in R$). Another possibility is to perform a statistical inference, deducing probabilities from a data sample representative of the expected input for the algorithm. A collection of datasets of real and synthetic images, covering most binary image processing application fields, is included in GRAPHGEN for this purpose.

The generation of an optimal decision tree is the first step of any optimization process provided by the framework.

This requires a general and flexible mean to describe the problem in a machine readable way. The modeling of a binary image processing problem with a decision table can be performed by feeding a regular function with all possible combinations of inputs and producing the corresponding actions, but other problems may require different and more complex strategies. Therefore, GRAPHGEN accepts different means to produce the list of rules: a YAML file can be used to define a task in terms of conditions and actions (Listing 1). Please refer to [31] for detailed examples of both use cases.

```
pixel_set:
   pixels:
      - {name: p, coords: [-1, -1]}
      - {name: x, coords: [ 0, 0]}
      - ...
   shifts: [1, 1]
conditions: [p, q, r, s, x]
actions: [nothing, x<-newlabel, x<-p, x<-q, x<-r, x
    <-p+r, x<-s, x<-r+s]
rules: [[1], [1], ..., [2], [3], [4], [3, 4], [5],
    ..., [3, 4, 5, 7]]
```

Listing 1. Example of YAML configuration file which defines the SAUF CCL algorithm [38] in GRAPHGEN. *pixel_set* identifies pixel *names*, their position inside the scanning mask and the mask *shift* size along x and y axes. *conditions* and *actions* represent respectively the list of conditions to check and actions to perform. For each rule, a set of equivalent actions is provided (*rules*).

2.2 State Prediction in Binary Image Processing

The second optimization step of GRAPHGEN, *prediction*, concerns the exploitation of the information gathered in the previous scan step, which can also be useful for the current one. Prediction can be used whenever some pixels are still part of the mask after the shift. In fact, if such pixels were checked in the previous step, there is no need to read them again. As an example, if we consider the 3×3 mask in Fig. 5, pixels from p_1 to p_6 may be used again after a unitary shift, but their identity will change accordingly. For example, if p_4 has been read, its value can be used instead of reading p_1 in the next step. This approach is commonly used with average/box filtering and running median [35]. He *et al.* [36] designed a CCL algorithm where the information provided by the values of already seen pixels is condensed in a configuration state, and the transition is modeled with a finite-state machine.

Grana *et al.* [37] proposed a paradigm to leverage already seen pixels, which combines prediction with decision trees. They noticed that the knowledge of pixels checked in the previous step could result in a simplification of the DTree for the current pixel. A reduced DTree can be computed for each possible set of known pixels, and then the trees can be connected to generate a single forest, which drives the execution of the algorithm.

In order to generalize the state prediction technique to every mask and shift, GRAPHGEN defines a standardized mask description structure and pixel naming, which allows the automatic tree reduction and forest generation.

2.2.1 Prediction of Already Accessed Pixels

The information about already known pixels is represented by a set of *constraints*, which are ordered pairs (p, v), where p is a pixel inside the mask, and v is its known value. A reduced tree is created from a more general one by applying the set of constraints: every node that contains a condition over a pixel included in the constraint set is substituted with the child corresponding to the known value. For example, if $(p, 1)$ is included in the constraint set, each node with condition p is replaced with its child on branch 1. The information gathered in each algorithm step is coded in the path from the DTree root to the selected leaf. In fact, already read pixels correspond to DTree nodes, and their values can be inferred from the chosen branches. Therefore, a constraint set is filled for each leaf of the general DTree, and is used to create a reduced version of it, meant to replace the complete tree in determining the action for the next pixel. Each reduced DTree is identified by an index, that is recorded in the leaf from which it was generated, in a field named *next*. This process creates a forest of reduced DTrees, which allows to apply state prediction to any algorithm. The complete DTree is only used for the first pixel of the row. Then, after a leaf has been reached and the proper action has been performed, the execution flow jumps to the root of the next reduced DTree associated to the leaf, and only reduced DTrees are used until the end of the row. The forest generated applying prediction on the DTree of Fig. 4 is reported in Fig. 6a.

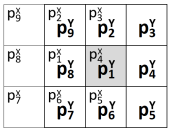


Fig. 5. Unitary horizontal shift for the 3×3 mask during image scan. Pixels named with "X" were inside the mask in the previous iteration while pixels named with "Y" are currently inside.

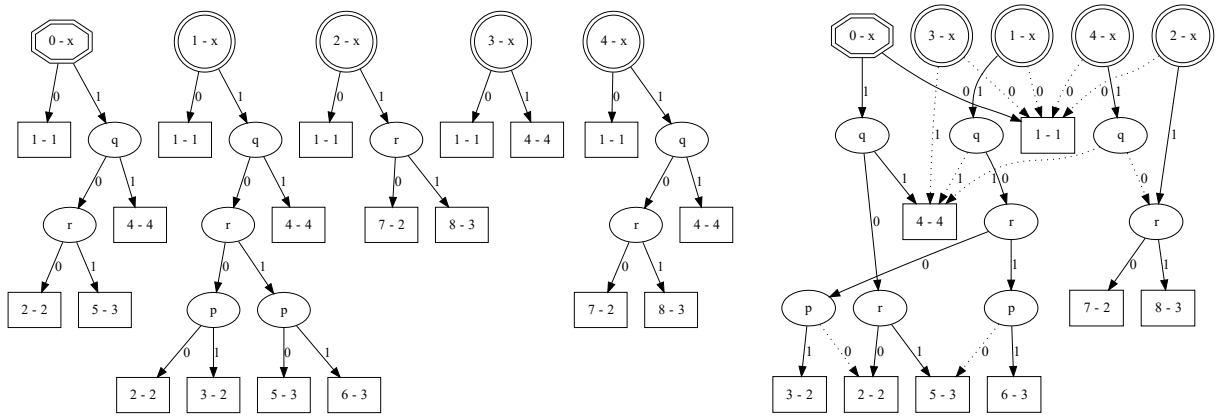
2.2.2 Prediction of External Pixels

It can happen that, at some point during the execution, the mask exceeds one or more borders of the image. In that situation, pixels outside the image are considered to have a fixed value out_v (usually 0). This observation leads to the construction of special constraint sets that are to be employed in specific areas of the image. For example, *first row* constraints set pixels in the upper part of the mask, which are outside the image when the first row is processed, to out_v. In the same way, *last row*, *first col*, *last col* constraint sets can be created, and when working on three dimensions, also *first slice* or *last slice*. The prediction of external pixels allows to avoid checks on pixel existence: in fact, every reduced DTree only considers pixels that are guaranteed to not exceed the borders of the input image. Thus, boundary checks can be removed.

2.3 From Trees to DRAGs

The ODTree generated in the first step of GRAPHGEN optimization procedure (Section 2.1) can contain identical or equivalent subtrees. These subtrees can be merged together, reducing the size of the compiled machine code.

The problem can be formalized as follows. The set of decision trees for the set of conditions C and actions A is



(a) DTree forest (b) DRAG

Fig. 6. (a) forest of DTrees obtained by applying state prediction on the ODTree of Fig. 4. In this example also the tree used for the first pixel of the row (tree to the left) is reduced considering external pixels constraints (Section 2.2.2). (b) is the DRAG originated from the compression of the forest (a). Leaves contain the action to be performed (left) and the index of the next tree/node (right). Root nodes are identified by an octagon (the starting one) or by circles (all the others) and have an index (left) plus the condition to be checked (right).

called $\mathcal{T}(C, A)$. $\mathcal{N}$ 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 $a(\ell) \in \mathcal{P}(A) \setminus \{\emptyset\}$, with $\ell \in \mathcal{L}$. Each node n has a left subtree $l(n)$ and a right subtree $r(n)$, each rooted in the corresponding child of n.

Definition 2 (Equal Decision Trees). Two decision trees $t_1, t_2 \in \mathcal{T}$, 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 $l(r_1)$ is *equal* to $l(r_2)$ and $r(r_1)$ is *equal* to $r(r_2)$.

Definition 3 (Equivalent Decision Trees). Two decision trees $t_1, t_2 \in \mathcal{T}$, having corresponding roots r_1 and r_2, are *equivalent* if either:

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

A pair of equal or equivalent trees can be merged into a single one, and both their parent nodes can point to it. The result of this transformation does not satisfy the definition of tree anymore, but it falls into the more common category of Directed Rooted Acyclic Graphs. As anticipated above, the conversion from tree to DRAG has the benefit of reducing the machine code size. The purpose is to make better use of the instruction cache, obtaining a more efficient code compression than that achievable by a compiler, which can merge identical pieces of code but cannot exploit equivalence between subtrees. This compression can be directly applied to an ODTree or to a decision forest obtained through state prediction. In the latter case, the merging procedure can involve subtrees of different trees, as long as corresponding leaves share the same *next* value.

The compression of a forest into a multi-rooted DRAG (Fig. 6b) is performed in two steps. The first step concerns the merging of equal subtrees. This is done by traversing the forest in any order, and merging each subtree with every equal one. The result of this operation is optimal and is always the same, whatever traversing order is chosen, because tree equality is a transitive relation [39]. Equivalent trees could be merged in a similar manner, taking the intersection of actions in the corresponding leaves. However, since tree equivalence is not transitive, this procedure would depend on the traversal order. Our aim is to find the optimum, which is the forest with the least nodes.

With respect to previous proposals [30], [40], our approach is a dynamic programming algorithm which guarantees to reach the optimal compression. In order to save computation time, we use a memoization technique that consists of a compact representation of trees in string form. The compression procedure starts creating a list of all the "stringized" subtrees in the forest that are equivalent to at least another tree. The list is sorted in descending order, so that larger trees come first. In recursive step n, the algorithm merges every couple of equivalent trees one at a time, and for each resulting forest continues the compression in step $n+1$. The recursion ends when no couple of equivalent trees remains.

This procedure ensures to find the compressed multi-rooted DAG with the minimum number of nodes. Moreover, sorting subtrees in decreasing order allows for a faster, greedy strategy, obtainable by stopping the procedure after it has reached the end of the recursion once. This is based on the heuristic assumption that it is better to merge larger subtrees first, which holds true in our experience: in all the examples that we tried, the best solution found by the algorithm is the first one.

2.4 Generating Algorithm Source Code

Regardless of the task addressed, any algorithm generated by GRAPHGEN scans the input image in a raster fashion. The processing starts at the beginning of each row with the corresponding *start tree* identified by index 0. The first operation performed is the increment of the column index,

Fig. 7. Sample images from the YACCLAB datasets. From left to right *3DPeS*, *Fingerprints*, *Hamlet*, *Medical*, *MIRflickr*, *Tobacco800*, *XDOCS*, *Hilbert*, *Mitochondria*, and *OASIS*.

```
for (int r = 1; r < rows; ++r) {
   int c = -1;
tree_0: // Start from tree 0
...
tree_2: // When processing tree 2
   c += 1; // Move to next pixel
   if (c >= cols - 1) {
      goto last_column_2; // Manage last column
   }
   if (CONDITION_X) {
      // Name this node, its needed later
      NODE_1:
      if (CONDITION_R) {
         ACTION_8 // Leaf: do action and
         goto cl_tree_3; // jump to next root
      }
      else {
...
tree_4: // When processing tree 4
   c += 1; // Move to next pixel
   if (c >= cols - 1) {
      goto last_column_4; // Manage last column
   }
   if (CONDITION_X) {
      if (CONDITION_Q) {
         ACTION_4 // Leaf: do action and
         goto cl_tree_4; // jump to next root
      }
      else {
        // Jump to existing subtree
        goto NODE_1;
      }
   }
...
}
```

Listing 2. Excerpt of the C++ code generated by GRAPHGEN for the DRAG of Fig. 6b. This example depicts the image scanning approach employed by GRAPHGEN-generated algorithms, the action performed in the leaves, the jump to the next tree, and the jump within conditions to reuse existing subtrees.

followed by an end-of-row check to decide whether a special *last column* tree should be used in place of the current one.

After processing one pixel, i.e., the traversal of the current tree is finished, a *goto* statement moves the execution flow to the beginning of the appropriate next tree, and the process continues for the next pixel. Then, after the current row has been entirely processed, the *for loop* moves the scanning process to the next one, until the whole image has been processed.

An excerpt of the C++ GRAPHGEN-generated code for the DRAG of Fig. 6b is provided in Listing 2 and exemplifies the whole process. As can be noticed, GRAPHGEN translates the concept of DTree and DRAG into running code as a bunch of nested if, $else$ and $goto$ statements, that leads the execution flow of an algorithm.

3 Three Showcase Applications

In this section, three use case applications of GRAPHGEN are presented and the algorithms generated by the framework are exhaustively evaluated in comparison with state-of-the-art solutions. The results discussed in the following have been obtained on a desktop computer running Windows 10 Pro (x64, build 10.0.18362) with an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz and an NVIDIA Quadro K2200 GPU using MSVC 19.15.26730 and CUDA 10.0.130 compiler (x64) with optimizations enabled.

All discussed algorithms (unless noted otherwise) use decision trees or forests generated by GRAPHGEN. They have been proved to be correct, i.e., the output result is exactly the one required by the given task.

The experiments are performed on the publicly available YACCLAB dataset [41], which covers most of the 2D and 3D applications of the analyzed tasks: video surveillance (2D-*3DPeS* [42]), fingerprints, (2D-*Fingerprints* [43]), medical (2D-*Medical* [44], 3D-OASIS [45], and 3D-*Mitochondria* [46]) and document (2D-*Tobacco800* [47], 2D-*Hamlet*, 2D-*XDOCS* [48]) analysis, real-world images (2D-*MIRflickr* [49]), and synthetic generated ones (3D-*Hilbert* curves). The datasets have highly variable resolution, density and amount of components. They originate from highly different sources, captured through various means (scans, photos, microscopy). Full description in [50] and example images in Fig. 7.

3.1 Connected Components Labeling

Connected Components Labeling aims at transforming an input binary image into a symbolic one, in which all pixels of the same object (connected component) are given the same label. The task has been originally introduced by Rosenfeld and Pfaltz [51] in 1966 and since then many papers designed algorithms to improve the efficiency of CCL [36], [40], [52], [53], [54], [55]. The CCL problem has a unique and exact solution, meaning that algorithms can use different strategies, but they must always provide the same output symbolic image, except for the specific label assigned to each connected component. The only difference among them is thus the time required to obtain the result.

For comparing the GRAPHGEN generated algorithms with existing ones, the open-source benchmarking system YACCLAB [41], [56], [57] has been used. It provides many state-of-the-art solutions, and allows to fairly compare the performance of CCL algorithms under various points of view.

The first significant improvement on CCL has been provided in [58] with the SAUF algorithm. The authors generated an optimal decision tree for the Rosenfeld mask

TABLE 1
Average run-time experimental results on 2D CCL algorithms in milliseconds. ASU is the Average Speed-Up over SAUF. The star identifies novel algorithmic solutions generated with the proposed techniques, all available in GRAPHGEN. Lower is better.

	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*	ASU
SAUF	0.885	0.377	6.900	3.194	0.373	10.611	41.369	1.000
PRED	0.865	0.312	6.297	2.831	0.326	10.126	38.535	1.103
PRED++*	0.866	0.312	6.299	2.831	0.326	10.127	38.531	1.103
BBDT	0.656	0.253	5.065	2.169	0.245	8.200	32.221	1.396
DRAG	0.650	0.253	5.019	2.177	0.247	8.121	32.185	1.399
Tagliatelle*	0.659	0.243	4.975	2.141	0.236	8.077	31.638	1.425
Spaghetti	0.612	0.234	4.766	2.055	0.226	7.702	30.435	1.492
Spaghetti$_F$*	**0.610**	**0.230**	**4.711**	**2.026**	**0.224**	**7.653**	**30.225**	1.507

(Fig. 1b) to reduce the average number of load/store operations during the scan of the input image. In [32] a subsequent major breakthrough was introduced (BBDT), consisting in a 2×2 block-based approach (Fig. 1c), again based on decision trees. In [36], He *et al.* demonstrated that it is possible to use a finite-state machine to summarize the value of pixels already inspected by the horizontally moving scan mask, and in [37] the authors combined the decision trees and configuration transitions in a decision forest generated from Rosenfeld mask (PRED). In [41], the conversion of a decision tree into a Directed Rooted Acyclic Graph (DRAG) has proved to be an effective technique to reduce the machine code footprint and lessen its impact on the instruction cache when considering large neighborhoods. Finally, the application of the state prediction approach to the 2×2 mask (Spaghetti) [30] further improved performance, setting the state-of-the-art on CCL.

Starting from the Rosenfeld mask (Fig. 1b), the generation of an ODTree (Fig. 4) recreates SAUF, the reference algorithm for the following Average SpeedUp (ASU) comparisons (Table 1). Then, the application of state prediction provides the PRED algorithm (Fig. 6a, ASU=1.103), and the final compression of the forest into a DRAG (Fig. 6b) produces PRED++. In this specific case, the effect of compression is not significant: because the PRED forest is already very small, reducing the code size does not affect instruction cache usage.

Tackling the problem with the Grana mask already proved to be an effective idea, and its GRAPHGEN-generated ODTree recreates BBDT (ASU=1.396), currently the default algorithm in OpenCV. Marginal improvements can be obtained by compressing this tree into a DRAG (ASU=1.399). Generating a prediction forest which is an uncompressed version of Spaghetti (we called this Tagliatelle) allows again to reduce the average number of memory accesses and conditional *ifs* when dealing with borders, while preserving the benefit of the block-based approach (ASU=1.425). Since the Tagliatelle tree is larger than that of BBDT, applying compression (Spaghetti) yields a bigger improvement (ASU=1.492). As explained in Section 2.1, our proposal is also able to consider image frequencies when generating the ODTree. Thanks to this, we are able to include them in the Spaghetti algorithm, further improving its speed, and thus outperforming the state-of-the-art with Spaghetti$_F$ (ASU=1.507). It is important to notice that frequencies are calculated in this case with the YACCLAB dataset, but it is also possible to compute them for specific problems, opening to further improvements.

TABLE 2
Average run-time experimental results of Thinning algorithms in milliseconds. The star identifies novel algorithmic solutions generated with the proposed techniques, all available in GRAPHGEN. Lower is better.

	Fingerprints	*Hamlet*	*Tobacco800*
GH	8.27	143.95	594.70
GH_LUT	3.60	72.89	296.85
GH_Tree*	2.62	48.70	192.11
GH_Spaghetti*	**2.39**	**47.08**	**186.59**
GH_Spaghetti$_F$*	2.43	50.97	206.59
ZS (OpenCV)	7.22	115.21	452.38
ZS_LUT	3.79	66.15	250.89
ZS_Tree	2.78	45.76	170.75
ZS_Spaghetti*	2.48	43.15	160.73
ZS_Spaghetti$_F$*	**2.45**	**42.98**	**159.88**
CH	5.78	119.75	452.77
CH_LUT	2.81	65.10	239.90
CH_Tree*	3.23	48.56	174.66
CH_Spaghetti*	1.99	48.02	173.55
CH_Spaghetti$_F$*	**1.95**	**47.78**	**172.63**

3.2 Image Skeletonization

Image skeletonization, another fundamental algorithm used in many computer vision and image processing tasks, aims at providing an approximate and compact representation of the objects inside images, reducing them to one pixel wide "skeletons". A common strategy to obtain it, called *thinning*, iteratively removes the outermost layers of connected components [59].

The algorithm proposed by Zhang and Suen (ZS) in [60] is one of the most famous and widely used, given its efficiency and simplicity. It is based on the 8-connectivity and exploits two sub-iterations performed alternatively to remove pixels. Chen and Hsu (CH) fixed some corner cases, and proposed a Look-Up Table (LUT) to speed up the process [61]. Furthermore, Guo and Hall [62] proposed a solution to better cope with 2×2 squares and diagonal lines, obtaining skeletons with less stair case artifacts. Besides the already mentioned LUT technique, iterations based on decision trees have been proposed in [63], to further speedup the ZS algorithm. These solutions have been proposed some decades ago, but are still commonly used [25], [26], [27] and included in many image processing libraries, such as OpenCV.

TABLE 3
Average run-time experimental results on chain-code algorithms in milliseconds. ASU is the Average Speed-Up over OpenCV. The star identifies novel algorithmic solutions generated with the proposed techniques, all available in GRAPHGEN. Lower is better.

	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*	ASU
Suzuki85 (OpenCV)	0.814	1.332	9.252	3.436	1.291	10.089	50.578	1.000
Cederberg_LUT	2.392	1.733	18.378	7.980	1.960	27.262	118.311	0.499
Cederberg_LUT_PRED	1.524	1.376	12.652	5.371	1.458	17.682	82.825	0.705
Cederberg_Tree*	0.613	1.092	6.749	2.950	1.136	7.534	47.545	1.231
Cederberg_Spaghetti*	0.596	1.052	6.535	2.728	1.069	7.307	46.079	1.284
Cederberg_Spaghetti$_F$*	**0.596**	**1.051**	**6.528**	**2.726**	**1.068**	**7.304**	**46.054**	1.286

Contrary to CCL, each thinning proposal provides different outputs and the choice depends on the application needs. ZS, CH, and GH algorithms (all using the mask of Fig. 1a) have been optimized with GRAPHGEN and compared with the open-source framework THeBE (THinning evaluation BEnchmark) [64]. Since the base algorithms produce different outputs, the comparison between execution times should be done only within each version of the single algorithm. For each technique, two variants have been manually implemented (naïve and LUT variant, both implementing basic prediction) and three others have been generated using GRAPHGEN: the Tree version only employs an ODTree, while the Spaghetti variants include state prediction and forest compression with DRAGs.

When comparing the various implementations within each algorithm (Table 2), LUT performs better than the base variant and Tree performs better than LUT by skipping unnecessary condition checks. Finally, Spaghetti further improves Tree by applying compression and prediction. Through image frequencies, the execution speed of the Spaghetti implementation can be slightly improved. Sometimes, however, frequencies slightly worsen the execution time, which can be attributed to the complex iterative nature of thinning: at every iteration, the distribution of patterns in the image changes, limiting the information gain of frequencies.

3.3 Contours Extraction

In a binary image, a contour is a sequence of foreground pixels that separates a connected component from the background. Several methods have been proposed for the representation of a contour, among which the chain-code is one of the most common. The first chain-code variation was proposed by Freeman [65]; it encodes the coordinates of one pixel belonging to the contour, and then follows the boundary, encoding the direction in which the next pixel shall be found.

Solutions to efficiently retrieve the chain-code from a binary image have been widely studied in literature. Sobel [66] proposed an algorithm that stores an 8-bit neighborhood code for each pixel, and performs a table-lookup to generate any neighborhood function. Suzuki and Abe [67] developed an extended border following algorithm, which discriminates between outer and hole borders, and also determines surroundness relations; this is the solution currently implemented in OpenCV. Zingaretti *et al.* [68] built a chain-code algorithm able to concurrently process all region boundaries in a single scan, providing a unified scheme for binary and gray-level images. Cederberg [69], in particular, proposed an alternative representation of chain-code, called Raster-scan Chain-code (RC-code), which can ease the retrieval when examining the image in a raster scan fashion, and presented an algorithm implementing this representation. Given its raster scan nature, which makes possible the application of state prediction, Cederberg algorithm has been chosen to undergo GRAPHGEN optimization.

In the RC-code, several coordinates are listed for each contour, and represent the first pixels that are hit in the border during the raster scan. Each of these pixels is called *max-point*, and is linked to two chains (R-chains), a left and a right one. A max-point can either be an *outer* max-point, when it is a transition from background to object, or an *inner* max-point, when at object-background transition. Every contour pixel met during the scan can either be a max-point (if it is not connected to any already known R-chain) or the next link of some existing R-chain. The same pixel can be a link for multiple R-chains; specifically, a border point that is a link for both a left and a right R-chain is called *min-point*, and determines the end of the two R-chains, which can then be merged. It also means that the left R-chain continues the same contour traced by the right R-chain, and therefore establishes an ordering between max-points of the same contour.

An RC-code is consequently composed of a list of max-points with their R-chains. The reconstruction of a contour starts from a max-point and follows its right R-chain until the end; then it follows, in reverse order, the connected left R-chain, and the process goes on until the starting max-point is met again. When computing the RC-code, is it sufficient to look at the mask of Fig. 1a to know the nature of the pixel, i.e., whether it is a max-point, a min-point or a chain link, and consequently know which action must be performed. With GRAPHGEN, a decision tree minimizing the average number of load/store operations needed to identify the state of a pixel has been generated. Then, prediction and code compression have been applied to this optimal decision tree.

In order to characterize the contour tracing performance, THeBE has been modified into BACCA (Benchmark Another Chain-Code Algorithm). The source code is available in [70]. The reference algorithm is the one implemented by OpenCV 3.4.7 [67], which uses an extremely optimized contour following approach, while the algorithm proposed by Cederberg [69] has been implemented in multiple variants: using lookup tables with and without basic state prediction (LUT_PRED and LUT), a version based on optimal decision trees and another one again with state prediction and compression (Spaghetti).

As can be observed in the experiments (Table 3), LUT implementations fail to compete with the carefully designed algorithm in OpenCV (ASU < 1). The GRAPHGEN-generated ODTree already provides a significant speedup (ASU=1.23), which is further improved by the Spaghetti versions (ASU=1.28).

The complexity of the algorithm requires a careful design of the decision table, since one pixel may be a min-point *and* a max-point *and* a chain continuation, requiring multiple actions to be performed in order. We thus encoded all possible cases in a bitmapped action number, which in turn selects the corresponding behavior.

3.4 What About 3D and GPUs?

Our proposal is not limited to 2D images and sequential CPU processing. While GPUs usually call for ad hoc massively parallel algorithms, we still evaluated trees generated by GRAPHGEN on a GPU implementation of CCL. Because of the parallel nature of GPU processing, state prediction is not feasible. Therefore, only the optimizations provided by the ODTree, its compression, and frequencies have been employed.

We compared the GPU-based BBDT and DRAG implementations to state-of-the-art CCL algorithms: Distanceless Label Propagation (DLP) by Cabaret *et al.* [71], Optimized Label Equivalence (OLE) by Kalentev *et al.* [72], Komura Equivalence[1] (KE) by Komura [74], Union Find[1] (UF) by Oliveira and Lotufo [75], Line Based Union Find (LBUF) by Yonehara and Aizawa [76], Block Equivalence (BE) by Zavalishin *et al.* [77]. Results are reported in Fig. 8. All GRAPHGEN-generated versions of BBDT and DRAG obtain a significant speed-up over KE, which has the lowest execution time among state-of-the-art algorithms. Since DRAG uses compression over BBDT, a slight advantage in run-time can be observed. Using image frequencies, DRAG_F achieves the best run-time over all algorithms, 18% faster than KE, the state-of-the-art approach.

The high amount of *if*- and *goto*-statements in the generated decision tree do not allow for efficient massively parallel and synchronized thread execution, but we have to consider the fact that branches depend on the pixel distribution in the mask. Neighboring pixels get processed concurrently, but they are not *i.i.d.*. Indeed, they are partially overlapped and definitely correlated. Thus, it is likely that most of the time, threads will traverse the same path through the tree/DRAG without causing any divergence. A similar behavior can be observed on other datasets (Fig. 10).

GRAPHGEN bases all of its processing on user-defined sets of rules and therefore enables to construct optimal decision trees and apply optimizations even on 3D-based algorithms. We report results only on 3D CCL, but our proposal can be easily applied to other 3D algorithms.

Due to the increased complexity that comes with 3D CCL, only algorithms using the Rosenfeld mask can be considered. For a block-based mask, the number of conditions and therefore the amount of different cases is too high to be computed within a reasonable time frame with current computing capabilities. For instance, a complete version of

1. Originally designed for 4-connectivity and later extended to 8-connectivity in [73].

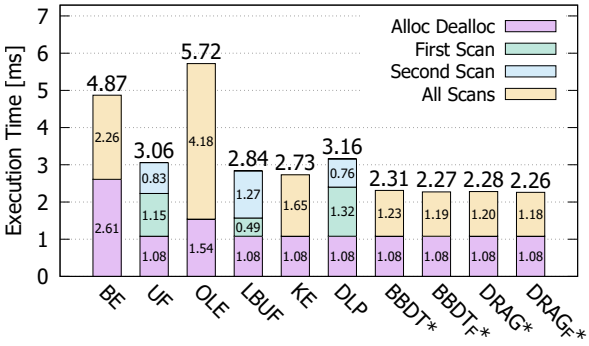
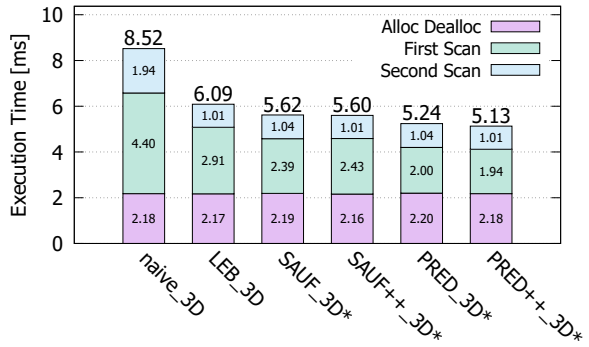
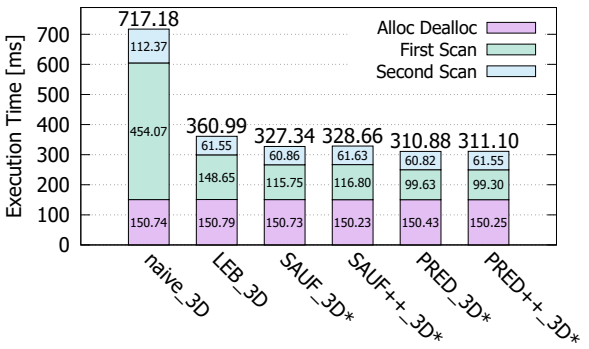


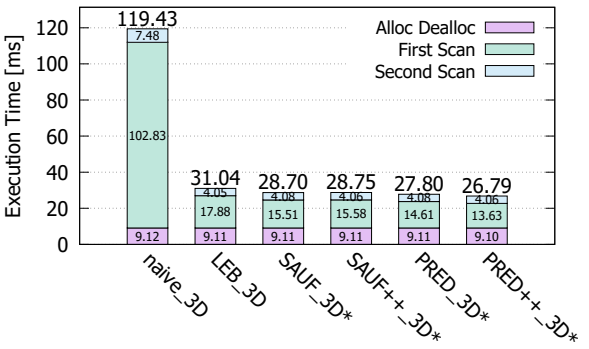
Fig. 8. Average run-time experimental results on 2D CCL algorithms on GPU on the *Hamlet* dataset in milliseconds. The star identifies novel algorithms generated with GRAPHGEN. Lower is better.



(a) Hilbert

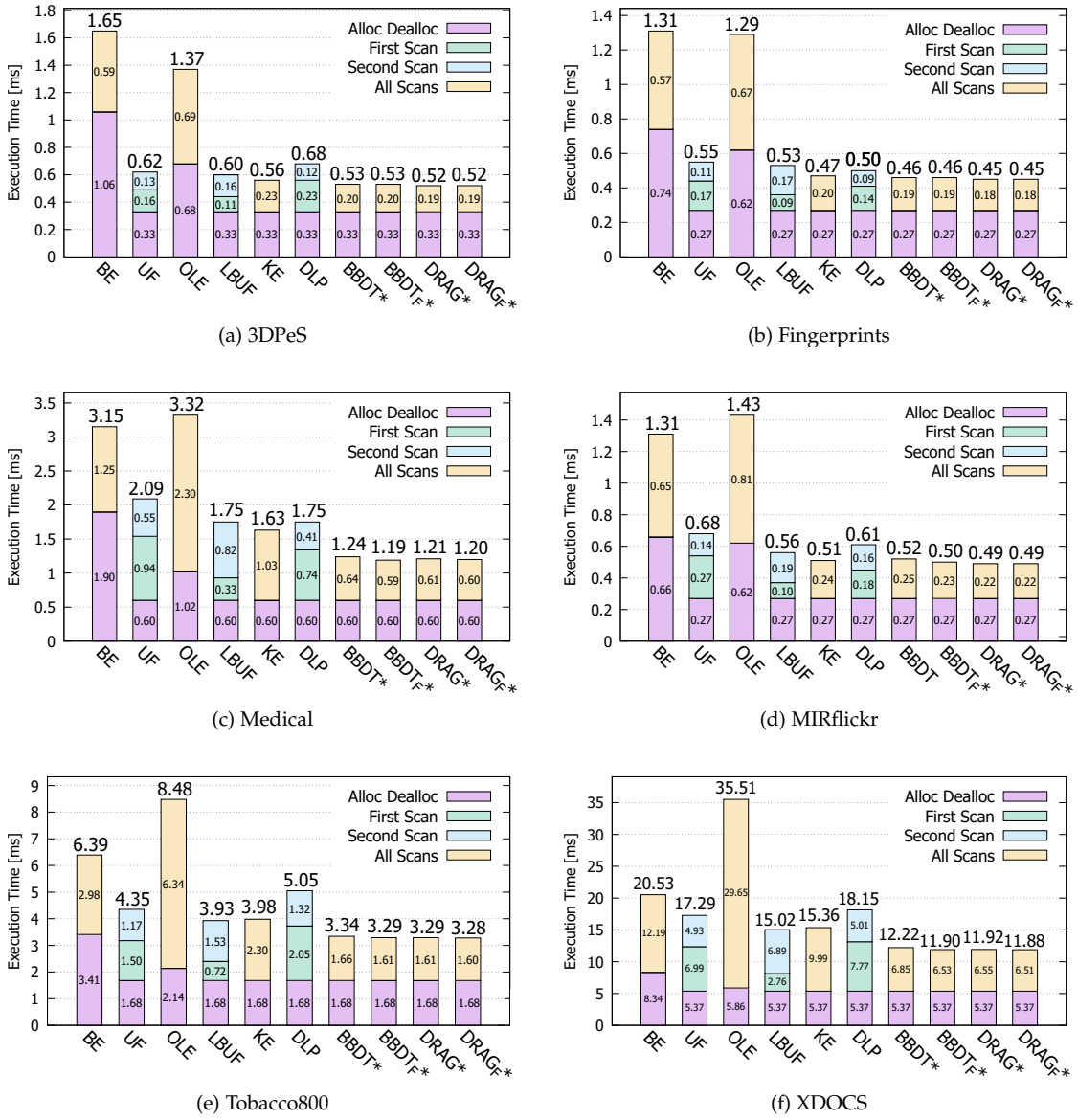


(b) Mitochondria



(c) OASIS

Fig. 9. Average run-time experimental results on 3D CCL algorithms in milliseconds. The star identifies novel algorithmic solutions generated with the proposed techniques, all available in GRAPHGEN. Lower is better.



(a) 3DPeS (b) Fingerprints

(c) Medical (d) MIRflickr

(e) Tobacco800 (f) XDOCS

Fig. 10. Average run-time experimental results on 2D CCL algorithms on GPU in milliseconds. The star identifies novel algorithms generated with GRAPHGEN. Lower is better.

the BBDT mask in 3D would contain 112 voxels requiring approximately $10^{24.68}TB$ of memory. The 3D version of the Rosenfeld mask contains 14 conditions: 9 voxels on the previous plane, 3 voxels in the previous row on the same plane, 1 voxel in the same row and same plane as x, and x itself.

The algorithms for 3D CCL generated with the strategy proposed in this paper are SAUF_3D, using the optimal decision tree, its DRAG-compressed version SAUF++_3D, PRED_3D, which adds state prediction, and its compressed version PRED++_3D. As a comparison, a naive 3D implementation of CCL that reads all neighbors and tries to merge all labels and the state-of-the-art for 3D CCL, Label-Equivalence-Based CCL by He *et al.* [78] (LEB) were benchmarked. LEB employs a handmade decision tree for the Rosenfeld 3D mask, built with a strategy that prioritizes pixels with the most neighbors. Fig. 9 shows the effectiveness of our proposal also in this case.

4 Conclusion

We presented a suite of algorithms that allows to generate optimal decision trees, and to automatically apply state prediction, compression and path-length optimization based on frequencies to any binary image processing problem. The only requirement for the user is to model his needs as a set of rules. The effectiveness of the proposed solution has been showcased on three different common binary image processes, covering both 2D and 3D scenarios. The GRAPHGEN-generated algorithms significantly improve the state-of-the-art. The source code of the proposed framework and its documentation are available in [31].

References

[1] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Springer, 2015, pp. 234–241.

[2] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[3] L. Canalini, F. Pollastri, F. Bolelli, M. Cancilla, S. Allegretti, and C. Grana, "Skin Lesion Segmentation Ensemble with Diverse Training Strategies," in *Computer Analysis of Images and Patterns*, vol. 11678. Springer, 2019, pp. 89–101.

[4] Y. Zhou, X. He, L. Huang, L. Liu, F. Zhu, S. Cui, and L. Shao, "Collaborative Learning of Semi-Supervised Segmentation and Classification for Medical Images," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 2079–2088.

[5] F. Pollastri, F. Bolelli, R. Paredes, and C. Grana, "Augmenting Data with GANs to Segment Melanoma Skin Lesions," *Multimedia Tools and Applications*, vol. 79, no. 21-22, pp. 15 575–15 592, 2019.

[6] L. Baraldi, C. Grana, and R. Cucchiara, "Hierarchical Boundary-Aware Neural Encoder for Video Captioning," in *2017 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1657–1666.

[7] F. Bolelli, L. Baraldi, and C. Grana, "A Hierarchical Quasi-Recurrent approach to Video Captioning," in *2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS)*. IEEE, 2018, pp. 162–167.

[8] B. Wang, L. Ma, W. Zhang, W. Jiang, J. Wang, and W. Liu, "Controllable Video Captioning with POS Sequence Guidance Based on Gated Fusion Network," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 2641–2650.

[9] C.-Y. Wu, C. Feichtenhofer, H. Fan, K. He, P. Krahenbuhl, and R. Girshick, "Long-Term Feature Banks for Detailed Video Understanding," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 284–293.

[10] X. Yang, X. Yang, M.-Y. Liu, F. Xiao, L. S. Davis, and J. Kautz, "STEP: Spatio-Temporal Progressive Learning for Video Action Detection," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 264–272.

[11] A. K. Bhunia, A. Das, A. K. Bhunia, P. S. R. Kishore, and P. P. Roy, "Handwriting recognition in low-resource scripts using adversarial learning*," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 4767–4776.

[12] T. Wilkinson, J. Lindstrom, and A. Brun, "Neural Ctrl-F: Segmentation-Free Query-By-String Word Spotting in Handwritten Manuscript Collections," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 4433–4442.

[13] I. H. Laradji, N. Rostamzadeh, P. O. Pinheiro, D. Vazquez, and M. Schmidt, "Where are the Blobs: Counting by Localization with Point Supervision," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 547–562.

[14] B. Li, W. Ouyang, L. Sheng, X. Zeng, and X. Wang, "GS3D: An Efficient 3D Object Detection Framework for Autonomous Driving," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 1019–1028.

[15] G. Máttyus, W. Luo, and R. Urtasun, "DeepRoadMapper: Extracting Road Topology from Aerial Images," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 3438–3446.

[16] A. Palazzi, D. Abati, F. Solera, R. Cucchiara *et al.*, "Predicting the Driver's Focus of Attention: the DR (eye) VE Project," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 7, pp. 1720–1733, 2018.

[17] M. Fabbri, F. Lanzi, S. Calderara, A. Palazzi, R. Vezzani, and R. Cucchiara, "Learning to Detect and Track Visible and Occluded Body Joints in a Virtual World," in *European Conference on Computer Vision (ECCV)*, 2018.

[18] E. Ristani, F. Solera, R. Zou, R. Cucchiara, and C. Tomasi, "Performance Measures and a Data Set for Multi-target, Multi-camera Tracking," in *Computer Vision – ECCV 2016 Workshops*. Springer, 2016, pp. 17–35.

[19] C. Liu, F. Wan, W. Ke, Z. Xiao, Y. Yao, X. Zhang, and Q. Ye, "Orthogonal Decomposition Network for Pixel-Wise Binary Classification," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 6064–6073.

[20] W. Chen and J. Hays, "SketchyGAN: Towards Diverse and Realistic Sketch to Image Synthesis," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 9416–9425.

[21] P. Tschandl, C. Rosendahl, and H. Kittler, "The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions," *Scientific data*, vol. 5, 2018.

[22] G. Yang, X. Song, C. Huang, Z. Deng, J. Shi, and B. Zhou, "DrivingStereo: A Large-Scale Dataset for Stereo Matching in Autonomous Driving Scenarios," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 899–908.

[23] T. Falk, D. Mai, R. Bensch, Ö. Çiçek, A. Abdulkadir, Y. Marrakchi, A. Böhm, J. Deubner, Z. Jäckel, K. Seiwald *et al.*, "U-Net: deep learning for cell counting, detection, and morphometry," *Nature Methods*, vol. 16, no. 1, pp. 67–70, 2019.

[24] F. Milletari, S.-A. Ahmadi, C. Kroll, A. Plate, V. Rozanski, J. Maiostre, J. Levin, O. Dietrich, B. Ertl-Wagner, K. Bötzel *et al.*, "Hough-CNN: Deep learning for segmentation of deep brain regions in MRI and ultrasound," *Computer Vision and Image Understanding*, vol. 164, pp. 92–102, 2017.

[25] J. Khodadoust and A. M. Khodadoust, "Fingerprint indexing based on minutiae pairs and convex core point," *Pattern Recognition*, vol. 67, pp. 110–126, 2017.

[26] F. Uslu and A. A. Bharath, "A recursive Bayesian approach to describe retinal vasculature geometry," *Pattern Recognition*, vol. 87, pp. 157–169, 2019.

[27] X. Wang, X. Jiang, and J. Ren, "Blood vessel segmentation from fundus image by a cascade classification framework," *Pattern Recognition*, vol. 88, pp. 331–341, 2019.

[28] S. Hannuna, M. Camplani, J. Hall, M. Mirmehdi, D. Damen, T. Burghardt, A. Paiement, and L. Tao, "DS-KCF: a real-time tracker for RGB-D data," *Journal of Real-Time Image Processing*, vol. 16, no. 5, pp. 1439–1458, Oct 2019.

[29] W.-C. Tu, S. He, Q. Yang, and S.-Y. Chien, "Real-Time Salient Object Detection with a Minimum Spanning Tree," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 2334–2342.

[30] F. Bolelli, S. Allegretti, L. Baraldi, and C. Grana, "Spaghetti Labeling: Directed Acyclic Graphs for Block-Based Connected Components Labeling," *IEEE Transactions on Image Processing*, vol. 29, no. 1, pp. 1999–2012, 2019.

[31] GRAPHGEN source sode. Accessed on 2020-12-15. [Online]. Available: https://github.com/prittt/GRAPHGEN

[32] 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.

[33] H. Schumacher and K. C. Sevcik, "The Synthetic Approach to Decision Table Conversion," *Communications of the ACM*, vol. 19, no. 6, pp. 343–351, Jun. 1976.

[34] 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.

[35] T. Huang, G. Yang, and G. Tang, "A Fast Two-Dimensional Median Filtering Algorithm," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 27, no. 1, pp. 13–18, 1979.

[36] 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.

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

[38] 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.

[39] 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.

[40] F. Bolelli, L. Baraldi, M. Cancilla, and C. Grana, "Connected Components Labeling on DRAGs," in *2018 24th International Conference on Pattern Recognition (ICPR)*. IEEE, 2018, pp. 121–126.

[41] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, "Towards reliable experiments on the performance of Connected Components Labeling algorithms," *Journal of Real-Time Image Processing*, vol. 17, no. 2, pp. 229–244, 2018.

[42] 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.

[43] D. Maltoni, D. Maio, A. Jain, and S. Prabhakar, *Handbook of Fingerprint Recognition*. Springer Science & Business Media, 2009.

[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] D. S. Marcus, A. F. Fotenos, J. G. Csernansky, J. C. Morris, and R. L. Buckner, "Open Access Series of Imaging Studies (OASIS): Longitudinal MRI Data in Nondemented and Demented Older Adults," *J. Cognitive Neurosci.*, vol. 22, no. 12, pp. 2677–2684, 2010.

[46] A. Lucchi, Y. Li, and P. Fua, "Learning for Structured Prediction Using Approximate Subgradient Descent with Working Sets," in *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 1987–1994.

[47] D. Lewis, G. Agam, S. Argamon, O. Frieder, D. Grossman, and J. Heard, "Building a test collection for complex document information processing," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2006, pp. 665–666.

[48] F. Bolelli, G. Borghi, and C. Grana, "XDOCS: An Application to Index Historical Documents," in *Italian Research Conference on Digital Libraries (IRCDL)*. Springer, 2018, pp. 151–162.

[49] 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.

[50] S. Allegretti, F. Bolelli, and C. Grana, "Optimized Block-Based Algorithms to Label Connected Components on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, pp. 423–438, 2019.

[51] A. Rosenfeld and J. L. Pfaltz, "Sequential Operations in Digital Picture Processing," *Journal of the ACM*, vol. 13, no. 4, pp. 471–494, 1966.

[52] 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.

[53] 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.

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

[55] 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.

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

[57] YACCLAB source code. Accessed on 2020-12-15. [Online]. Available: https://github.com/prittt/YACCLAB

[58] K. Wu, E. Otoo, and K. Suzuki, "Two Strategies to Speed up Connected Component Labeling Algorithms," *Pattern Analysis Application*, vol. 0, no. LBNL-59102, 2005.

[59] E. S. Deutsch, "Thinning Algorithms on Rectangular, Hexagonal, and Triangular Arrays," *Communications of the ACM*, vol. 15, no. 9, pp. 827–837, 1972.

[60] T. Zhang and C. Y. Suen, "A Fast Parallel Algorithm for Thinning Digital Patterns," *Communications of the ACM*, vol. 27, no. 3, pp. 236–239, 1984.

[61] Y.-S. Chen and W.-H. Hsu, "A modified fast parallel algorithm for thinning digital patterns," *Pattern Recognition Letters*, vol. 7, no. 2, pp. 99–106, 1988.

[62] Z. Guo and R. W. Hall, "Parallel Thinning with Two-Subiteration Algorithms," *Communications of the ACM*, vol. 32, no. 3, pp. 359–373, 1989.

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

[64] THeBE source code. Accessed on 2020-12-15. [Online]. Available: https://github.com/prittt/THeBE

[65] H. Freeman, "On the Encoding of Arbitrary Geometric Configurations," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 2, pp. 260–268, June 1961.

[66] I. Sobel, "Neighborhood Coding of Binary Images for Fast Contour Following and General Binary Array Processin," *Computer Graphics and Image Processing*, vol. 8, no. 1, pp. 127–135, 1978.

[67] S. Suzuki and K. Abe, "Topological Structural Analysis of Digitized Binary Images by Border Following," *Computer Vision, Graphics, and Image Processing*, vol. 30, no. 1, pp. 32–46, 1985.

[68] P. Zingaretti, M. Gasparroni, and L. Vecci, "Fast Chain Coding of Region Boundaries," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 4, pp. 407–415, 1998.

[69] R. L. Cederberg, "Chain-Link Coding and Segmentation for Raster Scan Devices," *Computer Graphics and Image Processing*, vol. 10, no. 3, pp. 224–234, 1979.

[70] BACCA source code. Accessed on 2020-12-15. [Online]. Available: https://github.com/prittt/BACCA

[71] L. Cabaret, L. Lacassagne, and D. Etiemble, "Distanceless Label Propagation: an Efficient Direct Connected Component Labeling Algorithm for GPUs," in *Seventh International Conference on Image Processing Theory, Tools and Applications*. IPTA, 11 2017.

[72] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider, "Connected component labeling on a 2D grid using CUDA," *Journal of Parallel and Distributed Computing*, vol. 71, no. 4, pp. 615–620, 2011.

[73] S. Allegretti, F. Bolelli, M. Cancilla, and C. Grana, "Optimizing GPU-Based Connected Components Labeling Algorithms," in *2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS)*. IEEE, 2018, pp. 175–180.

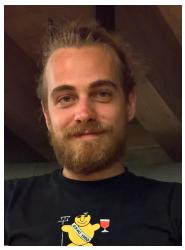
[74] Y. Komura, "GPU-based cluster-labeling algorithm without the use of conventional iteration: Application to the Swendsen–Wang multi-cluster spin flip algorithm," *Computer Physics Communications*, vol. 194, pp. 54–58, 2015.

[75] V. M. Oliveira and R. A. Lotufo, "A Study on Connected Components Labeling algorithms using GPUs," in *SIBGRAPI*, vol. 3, 2010, p. 4.

[76] K. Yonehara and K. Aizawa, "A Line-Based Connected Component Labeling Algorithm Using GPUs," in *2015 Third International Symposium on Computing and Networking (CANDAR)*. IEEE, 2015, pp. 341–345.

[77] S. Zavalishin, I. Safonov, Y. Bekhtin, and I. Kurilin, "Block Equivalence Algorithm for Labeling 2D and 3D Images on GPU," *Electronic Imaging*, vol. 2016, no. 2, pp. 1–7, 2016.

[78] L. He, Y. Chao, and K. Suzuki, "Two Efficient Label-Equivalence-Based Connected-Component Labeling Algorithms for 3-D Binary Images," *IEEE Transactions on Image Processing*, vol. 20, no. 8, pp. 2122–2134, 2011.



Federico Bolelli received the B.Sc. and M.Sc. degrees in Computer Engineering from Università degli Studi di Modena e Reggio Emilia, Italy. He pursued the Ph.D. degree from the same university where he is currently working as a postdoctoral researcher within the AImageLab group at Dipartimento di Ingegneria "Enzo Ferrari". His research interests include image processing, algorithms and optimization, medical imaging, deep learning, and historical document analysis.

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 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 and optimization, deep learning, and medical imaging.

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 Full 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 medical imaging, image processing, analysis of digital images of historical manuscripts and other cultural heritage resources, multimedia image and video retrieval, and color based applications. He published 5 book chapters, 38 papers on international peer-reviewed journals and more than 100 papers on international conferences.

TABLE 4
Average run-time experimental results on morphological operators *Erosion* and *Dilation* in milliseconds. ASU is the Average Speed-Up over OpenCV. The star identifies novel algorithmic solutions generated with the proposed techniques, all available in GRAPHGEN. Lower is better.

	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*	ASU
Dilation_OpenCV	0.797	0.282	5.774	2.554	0.365	9.884	35.783	1.000
Dilation_LUT	0.643	0.489	6.017	2.702	0.464	8.843	36.216	0.945
Dilation_Tree*	1.227	0.504	9.270	3.667	0.463	15.401	56.054	0.657
Dilation_Spaghetti*	**0.367**	**0.244**	**3.558**	**1.460**	**0.210**	**5.446**	**21.546**	**1.702**
Erosion_OpenCV	0.797	0.282	5.776	2.555	0.364	9.887	35.818	1.000
Erosion_LUT	0.677	0.480	6.185	2.797	0.463	9.181	36.476	0.922
Erosion_Tree*	0.393	0.299	4.041	2.108	0.378	5.948	24.374	1.387
Erosion_Spaghetti*	**0.243**	**0.206**	**2.870**	**1.275**	**0.202**	**4.122**	**16.981**	**2.139**

APPENDIX A

During the review process, we have been asked to add further experiments to prove that the proposed framework is not limited to the three showcase applications discussed in the paper. For this reason, a ubiquitous family of algorithms for binary image processing, i.e., mathematical morphology, has been selected. In particular, we focused on the two basic operations of *Erosion* and *Dilation* [79], because they represent the foundations for many other morphological operators. These algorithms require selecting a *structuring element*, so it is impossible to optimize all of the possible implementations with GRAPHGEN. However, following a common practice in computer vision libraries, the specialized algorithm for most frequent cases can be generated with our framework.

In OpenCV (version 3.4.7), the common 3×3 square structuring element has a specialized implementation. We thus select it as a candidate for optimization and apply GRAPHGEN to such a hard case, for which the DTree falls into a degenerate case, i.e., a list of conditions, giving alone no significant optimization.

The experimental results reported in Table 4 have been obtained with the same environment configuration described in Section 3. They show that the combination of prediction and compression strategies provided by GRAPHGEN allows for a significant improvement w.r.t. the performance of the OpenCV CPU implementation. In particular, the Spaghetti-like version reduces the total execution time providing an ASU of 1.7 for Dilation and 2.1 for Erosion, without resorting to hardware accelerators.

Given the importance of these two morphological operations, implementations leveraging the structuring element separability, applying SIMD or OpenCL vectorization and parallelization exist, leading to enormous performance improvements [80], [81]. However, the experiments reported in this Appendix demonstrate that further binary image processing algorithms could benefit from GRAPHGEN with extremely little design effort.

REFERENCES

[79] J. Serra and P. Soille, *Mathematical Morphology and Its Applications to Image Processing*. Springer Science & Business Media, 2012, vol. 2.
[80] M. Van Herk, "A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels," *Pattern Recognition Letters*, vol. 13, no. 7, pp. 517–521, 1992.
[81] J. Gil and M. Werman, "Computing 2-D min, median, and max filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, no. 5, pp. 504–507, 1993.