University of Modena and Reggio Emilia
XXXV cycle of the international doctorate school in
Information and Communication Technologies
Doctor of Philosophy dissertation in
Computer Engineering and Science

# Optimization of Connected Components Labeling Algorithms

Stefano Allegretti

Advisor: Prof. Costantino Grana
PhD School Director: Prof. Sonia Bergamaschi

Modena, 2023

# Abstract

Software optimization is the process by which an algorithm is made more efficient in terms of execution time, memory requirement or energy consumption. In the image processing field, time is in many cases the most restrictive constraint; optimization is therefore particularly important to allow tasks operating on considerable amounts of data to end in a reasonable time, and to consent real-time execution when required. An emblematic case in this sense is represented by binary images processing: connected components labeling, morphological operators, contour tracing. All these algorithms are commonly used in the pre- or post-processing phases of image analysis pipelines, even those based on modern deep learning techniques. This thesis introduces different methods for optimizing connected components labeling, for which innovative and efficient solutions are proposed in different contexts. By modeling the problem through a decision table, which defines the actions to be carried out for a pixel based on its neighborhood, and by applying different optimization techniques such as decision trees, block-based approach, state prediction and code compression, it is possible to describe an extremely efficient algorithm that represents the current state of the art. The same techniques are then applied to the labeling of 3D volumes, which represent a further challenge due to the growth of decision tables and the associated computational cost. Finally, specific solutions are explored for so-called bitonal images, stored with one bit per pixel, which allow to further increase efficiency through specific hardware instructions. Later, parallel solutions are explored for connected components labeling with GPUs. Algorithms based on the same decision trees used for the sequential counterpart are initially proposed, and then others are obtained by combining the block-based approach with pre-existing solutions, in order to establish the new state of

the art for massively parallel connected components labeling.

Review committee composed of:

Prof. Giuseppe Serra, Università degli Studi di Udine
Prof. Marco Bertini, Università degli Studi di Firenze

To my parents, Marco and Morena

# Contents

# List of Abbreviations

| | |
|---|---|
| **CNN** | Convolutional Neural Network |
| **CCL** | Connected Components Labeling |
| **GPU** | Graphic Processing Unit |
| **DT** | Decision Table |
| **SAUF** | Scan Array-based Union-Find (a CCL algorithm) |
| **YACCLAB** | Yet Another Connected Components Labeling Benchmark |
| **GRAPHGEN** | GRAPH GENerator |
| **THeBE** | THinning evaluation BEnchmark |
| **BACCA** | Benchmark Another Chain Code Algorithm |
| **DTree** | Decision Tree |
| **UF** | *Union-Find* (labels solving strategy and the associated tree-based data structure) - Union-Find (a GPU-based CCL algorithm) |
| **UFPC** | Union-Find with Path Compression (labels solving strategy) |
| **RemSP** | interleaved Rem's algorithm with SPlicing (labels solving strategy) |

| | |
|---|---|
| **TTA** | interleaved Rem's algorithm with SPlicing (labels solving strategy) |
| **BBDT** | Block Based with Decision Trees (a CCL algorithm) |
| **CT** | Contour Tracing approach (a CCL algorithm) |
| **CCIT** | a variation of BBDT algorithm (a CCL algorithm) |
| **LSL** | Light Speed Labeling (a CCL algorithm) |
| **SBLA** | Stripe-Based Labeling Algorithm (a CCL algorithm) |
| **PRED** | optimized state PREDiction (a CCL algorithm) |
| **CTB** | Configuration-Transition-Based (a CCL algorithm) |
| **CTBE** | Configuration-Transition-Based Extended/Enhanced (a CCL algorithm) |
| **CPU** | Central Processing Unit |
| **ODT** | Optimal Decision Tree |
| **LE** | Label Equivalence (a GPU-based CCL algorithm) |
| **KE** | Komura Equivalence (a GPU-based CCL algorithm) |
| **BE** | Block Equivalence (a GPU-based CCL algorithm) |
| **BKE** | Block-based Komura Equivalence (a GPU-based CCL algorithm) |
| **BUF** | Block-based Union-Find (a GPU-based CCL algorithm) |
| **LUT** | LookUp Table |
| **ASU** | Average Speed-Up |
| **OS** | Operating System |
| **GCC** | GNU Compiler Collection |
| **TDM-GCC** | Twilight Dragon Media - GNU Compiler Collection (a compiler suite for Microsoft Windows) |

**LBUF** Line-Based Union-Find

# Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisor Prof. Costantino Grana for many years of support, guidance, patience, motivation, enthusiasm, and knowledge. Being Costantino's student has been a privilege. He taught me to value high quality work, to think creatively and ask the right questions. I could not have imagined having better advisor and mentor for my PhD.

I also thank Dr. Eduardo Quiñones, my supervisor during the time I spent working as an intern at Barcelona Supercomputing Center, for his encouragement, passion and warmth.

I would like to thank the AImageLab research group and the Department of Engineering "Enzo Ferrari" for giving me the opportunity to be part of its graduate program and support me throughout. Thank you to my lab colleagues and friends Federico Bolelli, Michele Cancilla, Federico Pollastri, Laura Canalini and Marco Cipriano for their encouragement, for the insightful comments, for the stimulating discussions, for the days we were working together before deadlines, and for all the fun we have had in the last years.

I would like to thank all my teachers throughout the years for their dedication. None of my accomplishments would have been possible without the support of my family. Thank you to my parents Marco and Morena for nurturing me with passion for knowledge and supporting me throughout my life. Thank you to my sister Sara.

Finally, for all those I did not mention explicitly, but who I met during my journey and who have contributed in any way to make this possible: a heartfelt thanks.

# Chapter 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], perceiving and elaborating an interpretation of dynamic scenes [5, 6, 7, 8, 9], handwriting analysis and speech recognition [10, 11], surveillance, traffic monitoring and autonomous driving [12, 13, 14, 15], people tracking [16, 17], skeletonization [18], image synthesis [19] 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 data [15, 16, 20, 21], 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 [2, 12, 14, 16, 22] or to prepare training data [19]. When segmenting

images, a highly relevant task in medical imaging [1], *Connected Components Labeling* (CCL) is usually exploited together with voting strategies [23] to remove noise and produce the final segmentation map [2] or to count objects [12]. 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 [19], as in fingerprint analysis [24], vasculature geometry detection [25, 26], and road mapping [14].

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 [27, 28], 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.

Focusing on Connected Component Labeling, this thesis analyzes the state-of-the-art approaches proposed in the last decade, taking care how to fairly measure performance. We then introduce novel solutions to further optimize and improve state of the art of such kind of algorithms, showing how these optimization techniques can be generalized to boost the performance of any algorithm modeled with Decision Tables (DTs) [1].

We then introduce a novel framework that allows to automatically apply the most effective optimization strategies presented in this thesis and in literature in general to any problem modelled with Decision Tables. The framework, called GRAPHGEN (the all encompassing GRAPH GENerator), takes a definition of the problem as input, in terms of conditions that need to be checked and actions that have to be performed, and produces *C++* code, which implements the desired solution with all required optimizations as output.

We thus demonstrate the ability of the framework to automatically generate algorithmic solutions available in literature or presented in this thesis, apply these strategies to different problems, and even combine them to enhance performance and significantly improve state-of-the-art algorithms. To prove the generality of GRAPHGEN, the presented benchmarks are not limited to 2D sequential image processing algorithms, but also include 3D

[1] There is a large amount of algorithms that can be defined in such a way: Connected Component Labeling (CCL), Thinning, Chain Code, and Morphological operators are some of them. Generally, all those algorithms in which the output value for each image pixel is obtained from the value of the pixel itself and of some of its neighbors can be defined using decision tables.

and parallel GPU-based scenarios.

Then, specific solutions are explored for performing CCL on images stored with only 1 bit per pixel, also known as bitonal images. Two different kinds of solutions are proposed to tackle the problem: the first one makes use of GRAPHGEN-generated decision forests to label multiple pixels at once, while the second one exploits specific hardware instructions to maximize efficiency.

Finally, the task of CCL is considered in the field of GPU computing. After a detailed review of the state of the art, multiple novel solutions are proposed, inspired by the most successful techniques used for sequential algorithm: decision trees, and the block-based approach.

The most efficient CCL algorithms presented in this thesis represent, at the time of writing, the state of the art for both CPU and GPU, and are included in OpenCV, a common open source computer vision and machine learning software library. All the source code developed during the PhD is available on public repositories; URLs are listed in the specific Chapters.

# Activities Carried Out During the PhD

Beside the research activities described in this thesis, and those briefly summarized in Appendix A, I also took part in other teaching and service activities, which are reported below, together with a list of attended conferences and seminars.

## Participation to European Projects

- *H2020 - DEEPHEALTH:* a project funded under the call ICT-11-2018-2019 and recently terminated. The DeepHealth project provides HPC computing power at the service of biomedical applications; and applies Deep Learning (DL) techniques on large and complex biomedical datasets to support new and more efficient ways of diagnosis, monitoring and treatment of diseases. Among the other things, we are responsible for the development of the European Computer Vision Library (ECVL), one of the core elements of the project.

## Internship

- From March to August 2022, I have conducted an internship at Barcelona Supercomputing Center (BSC) under the supervision of Dr. Eduardo Quiñones. My project consisted in developing a deep learning tool to automatically detect and count bunch grapes in a vineyard, from frames acquired by a camera mounted on a car and processed in real time on an embedded system (Nvidia Jetson).

## Teaching Activities

- Teaching Assistant for the "Fundamentals of Computer Science I" undergraduate course, at Università degli Studi di Modena e Reggio Emilia (2019 and 2021-2022).

- Teaching Assistant for the "Fundamentals of Computer Science II" undergraduate course, at Università degli Studi di Modena e Reggio Emilia (2020-2022).

- Teaching Assistant for the "Operating System" undergraduate course, at Università degli Studi di Modena e Reggio Emilia (2020-2021).

## Grants and Awards

- Best paper award at the "18th International Conference on Computer Analysis of Images and Patterns" - CAIP.

## Conferences Attended

- IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), New Orleans, USA, 2022

- Joint 10th International Conference on Informatics, Electronics & Vision (ICIEV) and 5th International Conference on Imaging, Vision & Pattern Recognition (icIVPR), Kitakyushu, Japan, 2021

- 25th International Conference on Pattern Recognition (ICPR), Milano, Italy, 2021

- Joint 9th International Conference on Informatics, Electronics & Vision (ICIEV) and 4th International Conference on Imaging, Vision & Pattern Recognition (icIVPR), Kitakyushu, Japan, 2020
- International Conference on Image Analysis and Processing (ICIAP), Trento, Italy, 2019.
- IEEE International Conference on Image Processing, Applications and Systems (IPAS), Sophia-Antipolis, France 2018.

## Seminars Attended

- "Generative Flow Networks" —Prof. Yoshua Bengio (University of Montreal)—Harvard University, attended online, October 22, 2021.
- "Security and Quantum Technologies - Potential Uses and Risks in the Security Area" —Prof. Marco Pavone (Stanford University)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, October 14, 2021.
- "Brain-Inspired Computing Workshop" —Prof. Francesco Maria Puglisi (UNIMORE)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, October 07-08, 2021.
- "Digital Pathology: On the intersection of Computer Vision and Data Science" —Prof. Konstantinos N. Plataniotis (University of Toronto)— University of Toronto, attended online, June 29, 2021.
- "About Time" —Prof. Arnold Smeulders (UvA)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, September 30, 2020.
- "Academic English Workshop" —Dott. Silvia Cavalieri (UNIMORE)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, June 22 - July 21, 2020.
- "NVAITC Explained" —Frederic Pariente (NVIDIA)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, May 06, 2020.
- "Matematica Discreta" —Prof. Maria Rita Casali (UNIMORE)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, March 06 - May 20, 2020.

- "Data Science and Machine Learning: basics and applications to Health Care" —Prof. Paolo Missier (Newcastle University)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, November, 2019.

# Chapter 2

# Mathematical Preliminaries

This Chapter formalizes all the core elements related to the connected components labeling task, laying out the foundations for the rest of the discussion. Moreover, a common paradigm exploited by both sequential and parallel CCL algorithms is described: the *disjoint-set* data structure, also referred to as *union-find* or *merge-find*.

## 2.1 Basic Notations and Definitions

A binary image $I_d$ is a function defined over a $d$-dimensional rectangular lattice $\mathcal{L}_d$, where $I_d(x)$ is the value of pixel (voxel) $x \in \mathcal{L}_d$, with $x$ identified by its coordinates in the lattice, $(x_1, ...x_d)$. We are specifically interested in 2 and 3-dimensional images, the latter also called volumes.

The distance between pixels $x$ and $y$ can be defined in two different way, $D_1$ and $D_\infty$, also known as Manhattan and chessboard distance, respectively:

$$D_1(x, y) = \sum_{i=1}^{d} |x_i - y_i| \tag{2.1}$$

$$D_\infty(x, y) = \max_{i=1..d} |x_i - y_i| \tag{2.2}$$

Each distance has an associated neighborhood:

$$V_1^i(x) = \{y \mid D_1(x,y) \leq i\} \tag{2.3}$$

$$V_\infty^i(x) = \{y \mid D_\infty(x,y) \leq i\} \tag{2.4}$$

We can now use these generic definition to describe the different connectivities employed for 2D and 3D. In 2D images, there are two possibilities:

- *4-way connectivity.* Two pixels of the lattice, $p$ and $q$, are said to be 4-connected if they share a side. Formally, $\mathcal{N}_4(p) = V_1^1(p)$. We can say that $p$ and $q$ are 4-connected if $q \in \mathcal{N}_4(p)$, which implies $p \in \mathcal{N}_4(q)$. The name of the connectivity is intuitively derived from the number of sides a square has, four.

- *8-way connectivity.* If we also accept that pixels are connected when they share only a vertex, four more neighbours appear, obtaining the 8-way connectivity. Again, we can define $\mathcal{N}_8(p) = V_\infty^1(p)$ and say that $p$ and $q$ are 8-connected if $q \in \mathcal{N}_8(p)$, implying that $p \in \mathcal{N}_8(q)$.

Similar definitions can be provided for 3D volumes. In this context, pixels are often called voxels, and can be visualized as cubes; since two of them can share a face (with at most 6 different cubes), an edge (with at most 12 cubes) or a vertex (with at most 8 cubes), three different kinds of connectivity can be defined, i.e., 6-way, 18-way, and 26-way connectivity, whose formal definition is reported below:

$$\mathcal{N}_6(v) = V_1^1(v) \tag{2.5}$$

$$\mathcal{N}_{18}(v) = V_1^2(v) \cap V_\infty^1(v) \tag{2.6}$$

$$\mathcal{N}_{26}(v) = V_\infty^1(v) \tag{2.7}$$

If we want to define a mapping between 2D and 3D worlds, we can say that 4-way and 8-way connectivity on 2D images correspond respectively to 6-way and 26-way connectivity on 3D volumes. To simplify the discussion and ease the reading, in the following the the word *image* is used to refer to 3D volumes also. The generic symbols $\mathcal{L}$ and $I$ will be used, when suitable, for both dimensionalities.

The Connected components labeling task aims at identifying objects within a binary image, so it is mandatory to distinguish meaningful regions

from the rest of the image. The formers are usually called *foreground*, $\mathcal{F}$, while the latter is identified as *background*, $\mathcal{B}$.

It is a common convention to assign value 1 (or 255) to foreground pixels, and value 0 to background ones:

$$\mathcal{F} = \{p \in \mathcal{L} \mid \ I(p) = 1\} \tag{2.8}$$

$$\mathcal{B} = \{p \in \mathcal{L} \mid \ I(p) = 0\} \tag{2.9}$$

Connected components labeling aims at identifying disjoint objects composed of foreground pixels. Given a neighborhood definition $\mathcal{N}$ and two foreground pixels $p, q \in \mathcal{F}$, $\diamond$ is the *connectivity* relation defined as:

$$p \diamond q \Leftrightarrow \exists\{s_i \in \mathcal{F} \mid \ s_1 = p, s_{n+1} = q, s_{i+1} \in \mathcal{N}(s_i), i = 1, \ldots, n\} \tag{2.10}$$

Two pixels $p$ and $q$ are said to be *connected* when condition $p \diamond q$ is satisfied, meaning that a path of connected pixels from $p$ to $q$ exists or, in other words, you can move from $p$ to $q$ crossing only foreground pixels connected two by two. The proposed formalism does not consider background pixels, which are excluded from the concept of connectivity. $\diamond$ is an equivalence relation since *reflexivity*, *symmetry* and *transitivity* are satisfied by pixel connectivity. For this reason, we can define $[p]$, the equivalence class of a pixel $p$, as:

$$[p] = \{q \in \mathcal{F} \mid p \diamond q\} \tag{2.11}$$

Equivalence classes based on $\diamond$ relationship are called *Connected Components* (CCs). Every pair of connected components $[p]$ and $[q]$ is either equal or disjoint, meaning that the set of all connected components is a partition of $\mathcal{F}$.

Connected components labeling algorithms aim at assigning a different label to every connected component. When applied to $I$, the output of such an algorithm is a symbolic image $L$ where, for every $p \in \mathcal{F}$, $L(p)$ is the label of the connected component that $p$ belongs to ($[p]$), and for every $q \in \mathcal{B}$, $L(q) = 0$.

Depending on the chosen neighborhood definition, *n-neighborhood*, a connected components labeling algorithm is said to employ *n-connectivity*. Many computer vision tasks require 8-connectivity for 2D images, and 26-connectivity for 3D volumes. In fact, according to the *Law of Closure* of Gestalt psychology, our senses perceive an object as a whole even if it is composed of loosely connected parts [29].

## 2.2 The *Union-Find* Data Structure

Since two connected components are always disjoint, CCL can be seen as the partitioning of the lattice $\mathcal{L}$. A possible technique for performing such a partitioning consists of building initial sets of connected pixels, and then joining together sets that are part of the same connected component. This kind of problem can take advantage of the *union-find* data structure, that was firstly applied to CCL by Dillencourt *et al.* in [30].

The *union-find* data structure keeps track of $\mathcal{P}$, a partition of a set $\mathcal{S}$, and provides two basic operations on the elements of $\mathcal{S}$:

- `Find`($a$): returns the identifier of the subset that contains $a$ ($a \in \mathcal{S}$)
- `Union`($a$, $b$): joins the subsets containing $a$ and $b$ ($a, b \in \mathcal{S}$)

$\mathcal{P}$ is usually represented as a graph, in particular as a forest of directed rooted trees with orientation towards the root (*anti-arborescence*). Each element of $\mathcal{S}$, $a$, corresponds to a node and has a unique identifier $id_a$. An ordering exists between identifiers. A tree inside the forest identifies a subset belonging to $\mathcal{P}$.

A directed edge leading from $b$ to $a$, with $id_a < id_b$, states that $b$ and $a$ belong to the same subset. According to the definition of directed rooted tree, we will say that $a$ is the *parent* of $b$.

Of course, each element can have at most one outgoing edge. The *id* of a tree (subset) corresponds to that of its root node.

Representing $\mathcal{P}$ as a forest of trees is especially useful because a forest can be efficiently stored in memory using an array. Each index of this array is the *id* of a node, and the value stored at that index is the *id* of its parent node, or the index itself in the case of roots [31].

Fig. 2.1 provides an example representation of $\mathcal{P}$ as a forest of trees stored in memory as an array.

When applying *union-find* to GPU-based CCL algorithms, a common strategy is to identify pixels as nodes of the graph. In this case, the node *id* is the pixel raster index and each tree represents a connected component. An additional tree is required for background pixels. In such scenario the set $\mathcal{S}$ is the lattice $\mathcal{L}$. Thus, the array-based representation requires the same size as the output labels image: both require as many elements as the number of pixels in $\mathcal{L}$. Given that memory allocation on GPUs is considerably time consuming, since it requires an expensive operating

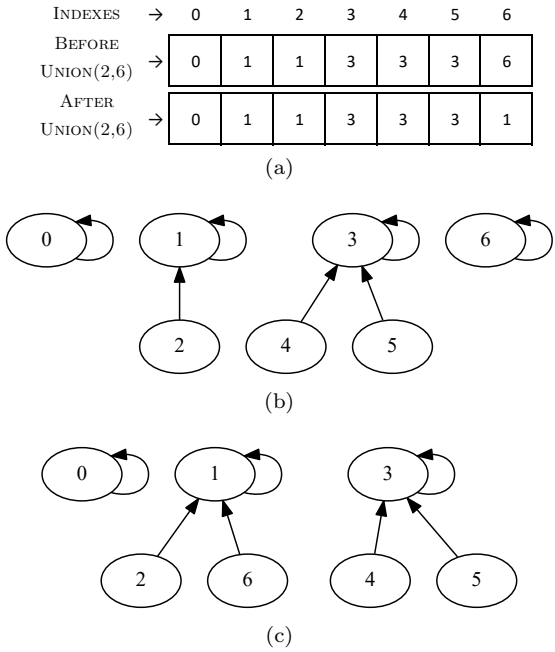



Figure 2.1: Visualization of $\mathcal{P}$ represented as a forest of trees and stored in memory as an array called $L$. (a) is $L$ before and after the execution of the `Union` operation between identifiers 2 and 6. (b) and (c) are the corresponding forests respectively before and after the `Union`. In this example, the `Find` function, called on 2 and 6 during the `Union`, returns 1 (the root of the tree 2 belong to) and 6 (since 6 is the root of the tree). The `Union` procedure replaces the identifier at index 6 of the $L$ array with 1, thus storing the equivalence between the two classes.

system call to the driver [32], many GPU CCL algorithms make the *union-find* array coincide with the output labels image $L$ [33, 34, 35, 36, 37].

A possible implementation of the *union-find* functions is reported in Algorithm 1. Implementations are made to fit CUDA data-parallel environment, where many threads can run the same function simultaneously, though on different input data. A complete description is reported in the following:

- `Find`($L$, $a$) consists of traversing the tree to which $a$ belongs, starting from $a$ and leading to the root node, whose index is the tree identifier.
- `Compress`($L$, $a$) is a procedure that links $a$ directly to the root of its tree. It is used to flatten *union-find* trees. When every tree in the *union-find* array exactly matches a connected component, the `Compress` procedure can be performed on every node/pixel to produce the final output image.
- `InlineCompress`($L$, $a$) is a variation of `Compress`, optimized for a data parallel environment. This procedure updates the father of $a$ at every step of the tree traversal. This way, possible concurrent threads that read $a$ can use the updated value. This approach is called *InlineCompression (IC)* and was firstly introduced in [34].
- `UnionNaive`($L$, $a$, $b$) first calls `Find` twice to get the roots of the trees containing $a$ and $b$, and then sets the smaller root as the father of the other one, thus joining the two trees into a single one. This solution does not take into account the possibility that two threads reach and modify the same root starting from different input nodes, possibly causing race hazards.
- `Union`($L$, $a$, $b$), firstly introduced by Oliveira *et al.* [33], solves the problems of `UnionNaive`, introducing CUDA atomic operations to make a thread aware of possible update losses caused by concurrent execution.

As said, when working on CPU the *union-find* forest is usually stored in memory using an array, because an array resides in consecutive memory locations. Moreover, two types of optimization techniques are commonly used to speed-up the naive approach for *union* and *find* operations: path compression and weighted *union* [38, 39]. The algorithm proposed by [31] only adopted the path compression strategy achieving a linear time complexity on average and under certain conditions.

Wu *et al.* [31] call the array that contains the equivalence information the $P$ array (short for parent links). According to [31], array $P$ can be filled as follows: every time a new provisional label is generated, array $P$ is extended by one element by use of assignment statement $P[l] \leftarrow l$. This operation adds a new single-node tree to the *union-find* trees. In other cases, a reference to $P[i]$ needs to be replaced by either a *find* or a *union*

operation. The implementations of such functions are the same reported in Algorithm 1: `Find` and `UnionNaive`.

Many other optimizations to solve the *union-find* problem have been proposed in literature. In [40], He *et al.* proposed the so called Three Tables Array (TTA). As the name suggests, this approach is based on a set of three arrays in order to link the sets of equivalent classes without the use of pointers. An *r_table* array contains information about the representative label of each class, a *n_label* array contains the index of the next equivalent label, thus providing a linked list structure, finally a *t_table* array contains the index of the last label of the list. This array-based structure turns out to be very effective, combining the performance of arrays with the benefits of a list-like structure in order to solve equivalences without scanning an entire array of equivalences.

Another classical strategy to solve this problem is the *Interleaved* algorithm, which differ from the *union-find* algorithms mentioned so far in that the two *find* operations in lines 14 and 15 of Algorithm 1 are performed as one interleaved operation. The aim is to void traversing portions of find paths. The first *Interleaved* algorithm provided in literature is Rem's algorithm (Rem) [41]. As originally presented, Rem integrates the *union* operation with a compression technique known as Splicing (SP) which aims at reducing paths during the execution. A detailed description of the *union-find* algorithms is available in [42].

**Algorithm 1** Possible implementation of *union-find* functions. $L$ is the *union-find* array, $a$ and $b$ are both array indexes and pixel identifiers.

```
1: function FIND(L, a)
2:     while L[a] ≠ a do
3:         a ← L[a]
4:     return a

5: procedure COMPRESS(L, a)
6:     L[a] ← Find(L, a)

7: procedure INLINECOMPRESS(L, a)
8:     id ← a
9:     while L[a] ≠ a do
10:        a ← L[a]
11:        L[id] ← a
12:    return a

13: procedure UNIONNAIVE(L, a, b)
14:    a ← Find(L, a)
15:    b ← Find(L, b)
16:    if a < b then
17:        L[b] ← a
18:    else if b < a then
19:        L[a] ← b

20: procedure UNION(L, a, b)
21:    done ← false
22:    while done = false do
23:        a ← Find(L, a)
24:        b ← Find(L, b)
25:        if a < b then
26:            old ← atomicMin(&L[b], a)
27:            done ← (old = b)
28:            b ← old
29:        else if b < a then
30:            old ← atomicMin(&L[a], b)
31:            done ← (old = a)
32:            a ← old
33:        else
34:            done ← true
```

# Chapter 3

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

Many approaches have optimized the computational load needed to label an image. In particular, the use of decision forests and state prediction have appeared as valuable strategies to improve performance. However, due to the overhead of the manual construction of prediction states and the size of the resulting machine code, the application of these strategies has been restricted to small masks, thus ignoring the benefit of using a block-based approach. In this Section, the block-based mask is combined with state prediction and code compression: the resulting algorithm is modeled as a Directed Rooted Acyclic Graph with multiple entry points, which is automatically generated without manual intervention, that we call Spaghetti. When tested on synthetic and real datasets, in comparison with optimized implementations of state-of-the-art algorithms, the proposed approach shows superior performance, surpassing the results obtained by all compared approaches in all settings.

## 3.1 Background

Connected Components Labeling (CCL) is a fundamental image processing algorithm that transforms an input binary image into a symbolic one in which all pixels of the same connected component (object) are given the same label. Introduced by Rosenfeld and Pfaltz [43], sequential CCL on binary images has been in use for more than 50 years in multiple image processing and computer vision tasks, including Object Tracking [44], Video Surveillance [45], Image Segmentation [46, 47], Medical Imaging Applications [48, 49, 50, 2], Document Restoration [51, 52], Graph Analysis [53, 54], and Environmental Applications [55].

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

Since 1966, many papers showed algorithms to improve the efficiency of CCL and, given the relevance of the task, this still represents a hot research topic [56, 57, 58, 59, 60, 61, 62].

Among others, a possible classification divides the algorithms into three main groups: raster scan, searching and label propagation, and contour tracing.

### 3.1.1 Raster Scan

*Raster Scan* algorithms scan the image exploiting a mask (see for instance Fig. 3.1) and solve equivalences between labels using different strategies. *Multiscans* approaches [63], for example, scan the image alternatively in forward and background directions to propagate labels until no changes occur in the output matrix. On the other hand, modern *Two Scan* algorithms [57, 29, 31, 64] solve equivalences between labels on-line during the first scan, usually storing them in a *union-find* tree. *Two Scan* algorithms are composed of three steps:

- *First scan*: scans the input image using a mask of already visited pixels, and assigns a temporary label to the current pixel/s, recording any equivalence between those found in the mask;
- *Flattening*: analyzes the registered equivalences and establishes the definitive labels to replace the provisional ones;

- *Second scan*: generates the output image replacing provisional with final labels.

When only statistics about connected components are required (*e.g.* area, perimeter, circularity, centroid), the second scan can be avoided, reducing the total execution time.

### 3.1.2 Searching and Label Propagation

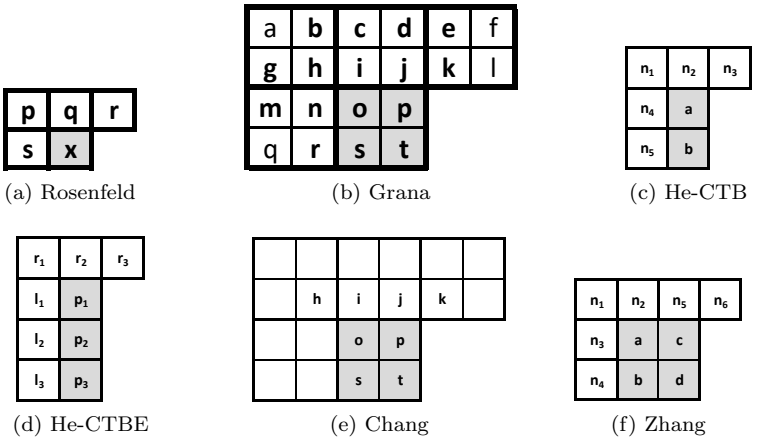
*Searching and Label Propagation* algorithms [65] scan the image until an unlabeled pixel is found: it receives a new label which is then repetitively propagated to all connected pixels. The process end when unlabeled pixels no longer exist. These algorithms scan the image in an irregular way.

### 3.1.3 Contour Tracing

*Contour Tracing* techniques [66] exploit a single raster scan over the image. During this process all pixels in both the contour and the immediately external background of an object are clockwise tagged in a single operation. Finally, the connected components have to be filled propagating contours' labels.

As regards sequential solutions, paper of the last fifteen years demonstrated the superiority of *two scans* algorithms over other approaches, based on multiple scans of the image or contour tracing.

The first proposal of a two scans algorithm was presented by Rosenfeld and Pfaltz in the sixties [43]. In the first scan, the provisional label for the current pixel $x$ is decided on the basis of the 4 already computed adjacent pixels out of the total 8, *i.e.* the three on the previous row and the one to the left of $x$. These pixel positions constitute the so called *Rosenfeld mask* (Fig. 3.1a). After this pioneering work, successive studies improved the efficiency of pixel neighborhood exploration and equivalences resolution. In particular, Samet and Tamminen [71] realized that the equivalence resolution problem can be described as a *disjoint-set union problem*, and therefore can be solved by means of the *union-find* algorithm [38]. The union-find algorithm, as introduced in Section 2.2, provides two basic operations over a collection of disjoint sets of equivalent labels: *find* retrieves the representative label of a set given an element belonging to it, and *union*




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

(d) He-CTBE (e) Chang (f) Zhang

Figure 3.1: Example of scan masks. Gray squares identify current pixels to be labeled using information extracted from white pixels. Rosenfeld mask (a) has been introduced in [43] and since then it has been used by several authors in many papers/algorithms [31, 64, 67]. Grana mask (b) introduces the concept of block-based scanning (*i.e.* scaning the image in 2×2 blocks). This approach has been used by many authors under different variations: (c) [57], (d) [68], (e) [69], and (f) [70].

joins two sets together. When a pixel is examined during the first scan, possible new equivalences between label sets are resolved online, with a *union*. Given its effectiveness and efficiency, union-find has been a staple in connected components labeling proposals thereafter, and authors proposed different implementations of it [31, 67].

For what concerns CCL performance optimization, the first significant improvement has been provided by Wu *et al.* [31], who proved an optimal strategy to reduce the average number of load/store operations during the scan of the input image, driven by Rosenfeld mask (Fig. 3.1a). They exploited a manually identified Decision Tree (DTree) to minimize the number of neighboring pixels to be visited in order to evaluate the label of the current one. This algorithm has been named SAUF (Scan Array-based Union-Find). Grana *et al.* [29] subsequently introduced a major breakthrough, consisting in a 2×2 block-based approach (Fig. 3.1b), which

modeled the CCL problem as a decision problem, and applied decision tables and trees to automatically generate the algorithm source code. They named this algorithm BBDT (Block-Based with Decison Trees).

Many improvements have been proposed since then [61], and few of them introduced significantly novel ideas, in particular:

- a proved algorithm to produce optimal decision trees [72];
- realizing that it is possible to use a finite state machine to summarize the value of pixels already inspected by the horizontally moving scan mask [57];
- combining decision trees and configuration transitions in a decision forest, in which each previous pattern allows to "predict" some of the current configuration pixels values, thus allowing automatic code generation (Section 3.2.2);
- switching from decision trees to Directed Rooted Acyclic Graphs (DRAGs), to reduce the machine code footprint and lessen its impact on the instruction cache (Section 3.2.3).

*Prediction*, as introduced by He *et al.* [73] and later improved in [57] and [68], has proven to be one of the most useful additions, as it allows to exploit already available information, save expensive load/store operations, and reduce execution time consequently. The idea behind prediction is tied to the fact that most existing algorithms scan the image and look at the neighborhood of a pixel through a mask. For each step of the scan process, an action is performed depending on the values of pixels inside the mask. When the mask is shifted along a row of the image it always contains some of the pixels it already contained in the previous step, though in different locations. If those pixels were indeed checked in the previous mask step, a second read of their value can be avoided by their removal from the decision process.

Anyway, in [57] the mask used was smaller than the one used in BBDT, because it was unfeasible to manually analyze all possible combinations produced by the prediction states. On the other hand, the procedure proposed in [74], as described later in Section 3.2.2 is suitable to be automatized, but still a small mask was employed. The reason, in this case, is that the larger the mask is, the more decision trees will populate the resulting forest, and the higher every tree will be. The machine code that

implements the algorithm resulting from the application of prediction to BBDT would be very large, and may have a negative impact on instruction cache. Therefore, despite load/store operations being less, the overall performance on real case datasets may be worse than that of the single tree variation. This was also observed in [68], an extension of [57]. For this reason, all works on prediction chose to avoid the complexity of the BBDT mask, and simplified it in various ways.

In this Chapter, we manage to combine the BBDT original mask and the *state prediction* paradigm by taking advantage of the code compression technique that converts a directed rooted tree into a DRAG [60]. The resulting process is modeled by a directed acyclic graph (DAG) with multiple entry points (roots), which correspond to the knowledge that can be inferred from the previous step. This guarantees a significant reduction of the machine code, which is better than that achievable by a compiler, since it can leverage the presence of equivalent actions in the trees leaves, and compress not only equal subtrees, but also equivalent ones.

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

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

Figure 3.2: *AND*-decision table for Rosenfeld mask. A different action for each condition outcome (mask configuration) is provided. To produce a more compact visualization the redundant logic have been reduced by means of the indifferent condition (represented by "–"). A condition marked with "–" does not affect the decision.

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

Figure 3.3: *OR*-decision table for Rosenfeld mask. When multiple foreground neighbors in the scan mask share equivalent labels, alternative actions can be performed to label the current pixel.

## 3.2 Preliminaries

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

### 3.2.1 Optimal Decision Trees

The procedure of collecting labels and solving equivalences is described in [29] as a *command execution metaphor*: the current and neighboring pixels in the mask provide a binary word, where 1 represents foreground pixels and 0 background. Each word represents a command that leads to the execution of a corresponding action, which can operate on pixels or over entire blocks with respect to the adopted mask. The possible actions are: *no action* if the current pixel/block is background, *new label* if it has no foreground neighbors, *assign* or *merge* based on the label of neighboring foreground pixels/blocks. In [75], Schutte showed that *decision tables* may conveniently be used to describe the relation between commands and corresponding actions. Additionally, Grana *et al.* [29] extended the classical concept of *AND*-decision tables to *OR*-decision tables. The former require all actions to be performed (*e.g.* perform action 3 *and* action 5 *and* ...), while the latter associate to every binary word (rule) a set of possible *equivalent* actions (*e.g.* perform action 1 *or* action 7 *or* ...)

An *OR*-decision table describes a distinctive characteristic of the CCL problem: when multiple foreground neighbors in the scan mask share equivalent labels, alternative actions can be performed to label the current pixel or block. Considering the Ronsenfeld mask (Fig. 3.1a), the associated *AND*/*OR*-decision tables are respectively reported in Fig. 3.2 and Fig. 3.3.

The *AND*-decision table can be converted into an Optimal Decision

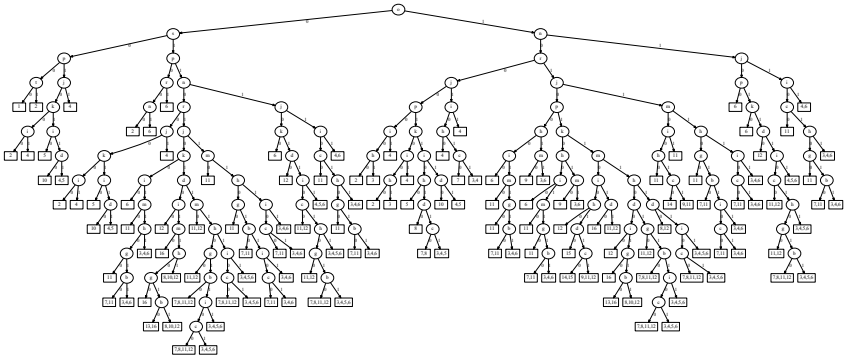


Figure 3.4: Optimal DTree obtained using the algorithm presented in [72] and the mask of Fig. 3.1b. Best viewed online.

Tree (ODT) through the use of the dynamic programming approach introduced by Schumacher *et al.* [76]. The process described by Schumacher ensures to obtain a DTree that minimizes the average number of conditions to check when choosing the correct action to be performed. This strategy has been then extended to *OR*-decision tables by Grana *et al.* in [72], as we already mentioned. In that paper, the authors prove the optimality of the conversion from *OR*-decision tables to DTree and provide a mechanism to automatically convert a DTree into running code. It may happen that, after the optimization, a leaf still contains equivalent actions, so the authors suggest that a random one can be chosen without affecting the result. This automatic procedure allows to extend the algorithm to complex masks, as the one in Fig. 3.1b. This mask allows the labeling of four pixels ($o$, $p$, $s$ and $t$) at the same time, thus reducing the number of load/store and merge operations required. Indeed, labels equivalence is automatically solved within $2 \times 2$ blocks without requiring additional merges.

The ODT obtained with the aforementioned strategy and using Grana mask is reported in Fig. 3.4. Here, the total number of nodes is 136 and leaves are 137. Differently from what previously presented in [72], this tree still shows all the equivalent actions in its leaves, which will be exploited later for further optimizations.

### 3.2.2 State Prediction

He *et al.* [73] were the first to realize that, when the mask shifts horizontally through the image, it contains some pixels that were already inside the mask in the previous iteration.

Considering the Rosenfeld scan mask, it can be observed that pixel $x$ will be the next $s$, $q$ will be the next $p$, and pixel $r$ will be the next $q$. See Fig. 3.5 as a reference. A similar observation can be drawn for the other masks. Then, in the case those pixels were indeed checked in the previous step, a repeated read can be avoided. He *et al.* [57] addressed this problem condensing the information provided by the values of already seen pixels in a configuration state, and modeled the transition with a finite state machine. The algorithm was designed for the specific task, and no provision of a general methodology is provided in the paper.

Grana *et al.* [74], instead, proposed a general paradigm to leverage already seen pixels, which combines configuration transitions with the decision trees. Algorithms that make use of a DTree usually traverse the same tree for each pixel of the input image. In that paper, authors noticed that the exploitation of values seen in the previous iteration could result in a simplification of the decision tree for the current pixel. A reduced DTree can be computed for each possible set of known pixels. Then, trees can be connected into a single graph (forest), which drives the execution of the CCL algorithm on the whole image. It is noteworthy that, for a certain position of the mask, the set of pixels that theoretically do not need to be read are not only the already seen ones, but also pixels that are outside the input image, which are usually considered background. In fact, Grana *et al.* also exploited the information about the position of the mask in the image, and built special reduced DTrees that are only used in correspondence with borders. The whole optimization strategy has been

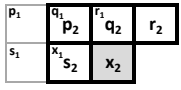



Figure 3.5: Unitary horizontal shift for Rosenfeld mask during image scan. Pixels named with "1" were inside the mask in the previous iteration while pixels named with "2" are currently inside the mask: $q_1 \rightarrow p_2$, $r_1 \rightarrow q_2$, and $x_1 \rightarrow s_2$.

applied *by hand* to the SAUF algorithms, *i.e.* using the Rosenfeld mask.

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

### 3.2.3 From Trees to DRAGs

In [60], authors noticed the existence of identical and equivalent subtrees in the optimal decision tree obtained using the algorithm by Grana *et al.* [72]. They observed that identical subtrees were merged together by the compiler optimizer, with the introduction of jumps in machine code. The result of such a merging is the conversion of a tree into a Directed Rooted Acyclic Graph, which they called DRAG.

While the code compression operated by the compiler optimizer is aimed at the reduction of code footprint, the compiler is only capable of recognizing identical pieces of code. In [60], this optimization is enhanced by merging not only identical subtrees, but also equivalent ones. The formal statement of the problem is as follows.

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

**Definition 3.2.3.1** (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 $\mathscr{r}(r_1)$ is *equal* to $\mathscr{r}(r_2)$.

**Definition 3.2.3.2** (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 $\not\in(r_1)$ is *equivalent* to $\not\in(r_2)$.

A pair of equal or equivalent trees can be merged into a single one, to which both their parents can point. The aim of the conversion of a decision tree to a DRAG is the exploitation of such merging operations, in order to minimize the total number of nodes. One possibility consists of traversing the tree in a chosen order, and merging each subtree with every possible equal one. Since $\mathcal{DT}$ equality is a transitive relation, the result of this operation does not depend on the order chosen for traversing the tree [77].

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

## 3.3 Outline of the Spaghetti Algorithm

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

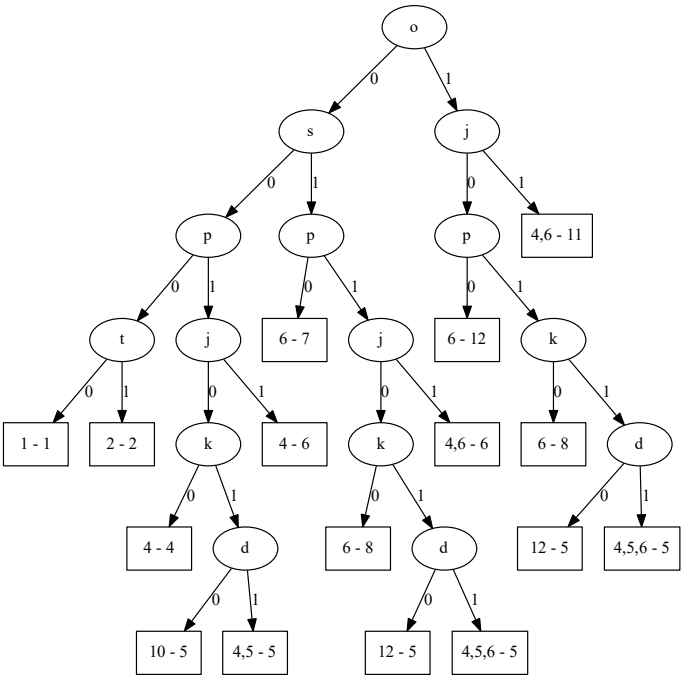



Figure 3.6: Reduced tree obtained from the optimal DTree (Fig. 3.4) through the application of the set of constraints $\{(h,0),(n,1),(i,1)\}$. Leaves contain equivalent actions to perform (to the left) and the index of the next tree to use (to the right) separated by a dash.

### 3.3.1 State Prediction with Grana Mask

In the previous work by Grana *et al.* [74], the Rosenfeld mask is adopted, and both the forest and the graph that links trees together are manually built. In order to implement the same approach with a different and more complex mask, such as the BBDT one, we employ a generic algorithm that automatically generates forests of reduced DTrees and connects them into graphs. This approach is described in the following of this Section.

**Forest of Reduced Trees.** We represent the information about an already known pixel through a *constraint*, which is an ordered pair $(p, v)$,

where $p$ is a pixel of the mask, and $v$ is its value. For each leaf of the complete tree, we build a set of constraints that stores the information about pixels that have been read in the path to the leaf. These constraints are modified according to the mask shift. Considering for example the simplified case with unitary shift in Fig. 3.5, a constraint over the pixel $x$ will turn into a constraint on pixel $s$. For each leaf, a reduced tree is then generated through the application of the corresponding set of constraints to the original tree.

A reduction is performed traversing the original tree, from root to leaves, recursively. Each node that contains a condition over a pixel included in the constraints set is substituted with its child that corresponds to the constraint value. This reduction, that is a pruning of the original tree, allows to remove the checking of already known pixels. As an example, if a constraints set contains $(i, 0)$, each node of the original tree with condition $i$ is replaced with its child associated to branch 0. Every reduced tree is given a unique index.

Obviously, after the reduction it is possible for a node of a reduced tree to have two identical branches. Those can therefore be merged together, by substituting the parent node with one of its equal children. Then, possible pairs of identical reduced trees are looked for, and duplicates are deleted. An example of a reduced tree is shown in Fig. 3.6.

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

**Beginning of Rows.** At the beginning of the row, no information is available about previously seen pixels. Thus, the complete decision tree could be used. Anyway, we know that some pixels of the mask are out of the borders of the input image. For the BBDT mask, the case is shown in Fig. 3.7. Since those pixels are always considered to be background, there is no need to use the complete tree. Conversely, a reduced tree to be used at the beginning of the row is built, imposing constraints that set to 0 every pixel outside the image. We call this tree *start tree*. The start tree is added to the main forest, and is considered a main tree to all effects.

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

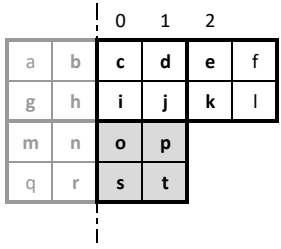



Figure 3.7: Grana mask configuration at the begin of a line. Dotted line represents the left border of the image.

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

The second problem is tied to the specific mask chosen by the CCL algorithm. When using BBDT mask, we notice that end of row constraints depend on whether the number of columns of the input image is even or odd (Fig. 3.8).

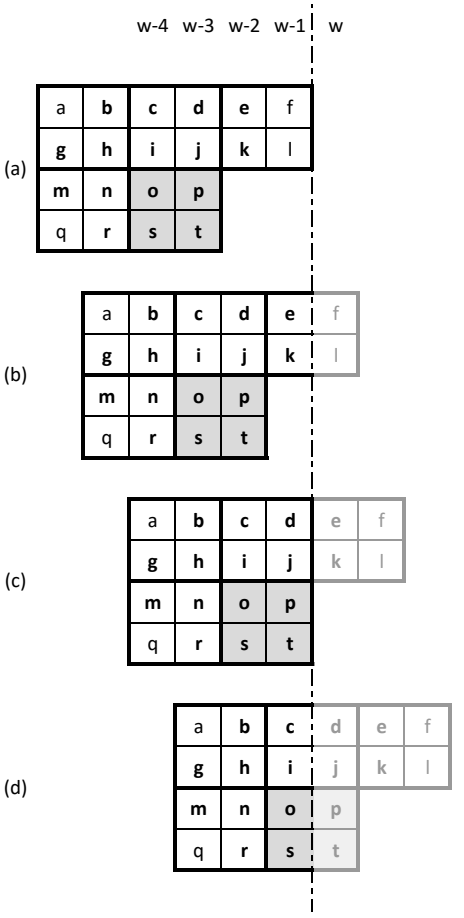



Figure 3.8: Possible configurations of Grana mask when it reaches the end of a line. The dotted line represents the right border of the image. Configurations (a) and (c) will occur when image has an even number of columns. Configurations (b) and (d) are required with odd number of columns.

Thus, for each main tree, two different end trees are generated, one to be used when the columns are odd and one for when they are even. The *end forest* is therefore divided into two disjoint sets, that we will call *end forest even* and *end forest odd*. The merging of identical branches and removal of duplicate trees are performed on end trees too.

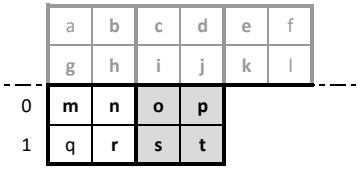



Figure 3.9: Grana mask configuration at the first line. The dotted line represents the upper border of the image.

**First Row and Last Row.** It is also possible to observe that when the first row is scanned, the upper part of the mask is outside of the image (Fig. 3.9). Similarly, if the number or rows is odd, when the last row is scanned the bottom line of the mask exceeds the lower border of the image. This observation leads to the construction of *first row forests* and *last row forests*, which can be obtained from the *main forest* and the *end forest* through the application of first line constraints or last line constraints.

Finally, in the far-fetched but possible case that the input image has only one row, a special set of forests is needed in which both first line and end line constraints are considered. Those were created and included in the algorithm, for such uncommon situations. The way in which different forests alternate during the scan of an image is depicted in Fig. 3.10.

### 3.3.2 DRAGs

Since main and end forests described in the previous Sections have a very large number of nodes, we exploit the merging of equal and equivalent subtrees in order to reduce code footprint and make a better use of the instruction cache. Differently from the original work by Bolelli *et al.* [60], however, we do not have a single tree, but three whole forests for each line case (*i.e.* first row, last row, and middle rows). In fact, equivalent subtrees can be merged together even if they belong to different trees. This approach leads to the conversion of a forest into a DAG with multiple entry points (roots). Another peculiarity of our specific case is that leaves of main trees do not only contain actions, but also a link to the root of the

subsequent tree to use after the mask shift.

Thus, in order for two leaves to be considered equal or equivalent, it is also required that they point to the same next tree. A consequence is that a subtree of a main tree will never have an equivalent one in the end forests, since end trees leaves do not have pointers to other trees. Anyway, it is possible for a subtree in the end forest even to have an equivalent one in the end forest odd. A choice had to be made on whether to reduce the two end forests jointly or separately. Theoretically, this choice should not impact instruction cache, because only one end forest is needed for a certain image, so pieces of code that implement trees belonging to different end forests will never conflict for a cache line during the labeling of an image. We tried both the possibilities anyway, and experimental tests did not show performance discrepancies. In the following, the two end forests will be considered separately, but every operation can be adapted to the joint option by simply substituting the two sets of end trees with a single set that represents their union.

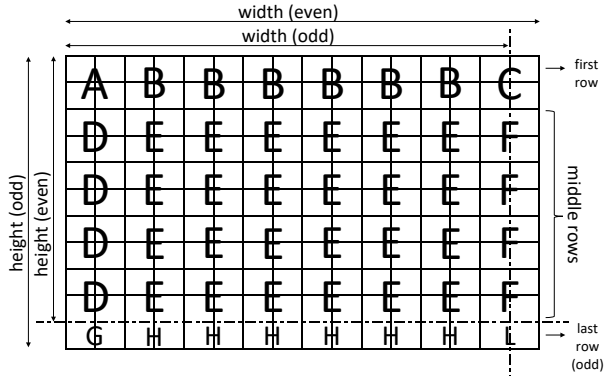
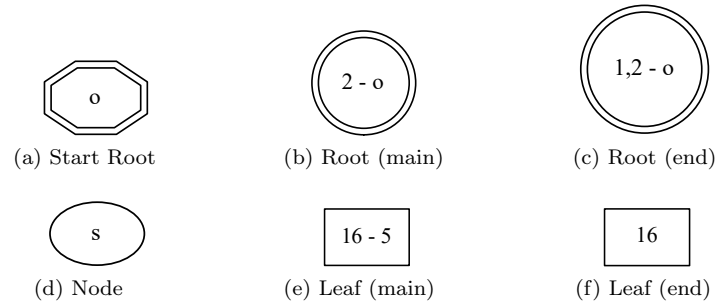



Figure 3.10: This figure shows which trees/forests should be used in each part of the image. For what concerns first row, “A” is the start tree, “B” represents the main forest and “C” is the end forest which automatically handle the odd/even number of cols. “D”, “E”, “F” have a similar meaning but for middle rows. When image has an odd number of rows an additional group of forests is required: the end line forests. In that case “G” represents the last row start tree, “H” is the main forest and “L” is the end forest.

The conversion of a forest to a DAG with multiple roots consists of two steps. The first one is the merging of equal subtrees, as described in Section 3.2.3. The sole difference is that we do not traverse a tree only, but a set of trees one by one, and for each subtree we search for equal ones in the whole forest. This operation is guaranteed to be optimal, because it does not depend on the order in which subtrees are traversed. The second step involves equivalent subtrees reduction. Similarly as described in Section 3.2.3, if during the tree traversal, equivalent subtrees are merged together as soon as they are found, there is the risk to change the list of equivalent actions so that now it is impossible to perform another substitution of larger subtrees. The exhaustive optimal solution adopted by Bolelli *et al.* in [60] is not applicable to our case, because the amount of equivalent forests is too large for the execution time to be reasonable.




(a) Start Root (b) Root (main) (c) Root (end)

(d) Node (e) Leaf (main) (f) Leaf (end)

Figure 3.11: (a) is the root of a start tree with the corresponding condition to check (*o* in this example). (b) and (c) are the symbols used for the roots of a generic tree inside the forest: in (b) are specified the index of the tree (2 in this example) and the condition to be checked (*o* in this example), in (c) the *group* to which the tree belongs is also reported (1 in this example). See Section 3.3.2 for details about tree *groups*. Nodes of the tree, and associated conditions to check, are represented as in (d). (e) and (f) are the symbols for leaves: the first one contains both the action to be performed (16 in this example) and the index of the next tree (5 in this example), the second one contains only the action. (b) and (c) are used inside the main forest, while (c) and (f) are required to depict end line forests.

To solve the problem, we use a heuristic greedy algorithm which tries to prioritize more valuable substitutions, meaning that larger equivalent subtrees get substituted before smaller ones. This is heuristic since there is no guarantee that many smaller substitutions could be performed if the large one had not been performed, but we could not spot any counter example in practice. The process follows these steps:

1. all trees in the forests are traversed and each of them is unrolled into a string with a memoizing strategy, along with a list of the possible actions and the next tree reference. Each element has also got a pointer to the subtree it represents;

2. the list of "stringized" subtrees is heuristically sorted by string length (prefer larger trees) and then lexicographically;

3. now we can move through the list and remove all the elements whose strings do not appear more than once, since no subtree shares the same conditions. This shortened list will contain possible substitutions;

4. going through the list, we now check if two entries with the same string have a non empty actions lists intersection. In this case, one of the subtrees can be replaced by a pointer to the other after intersecting the actions lists;

5. the process starts again after the removal, because now the graph structure is different and some of the smaller equivalences have already been resolved. The memoization step is not required again, since the strings are the same as before, but the list is recreated with the now reduced structure.

The process ends when no substitution is possible. The main and the end forests, specific for middle rows and converted to DAGs, are depicted respectively in Fig. 3.12 and Fig. 3.13.

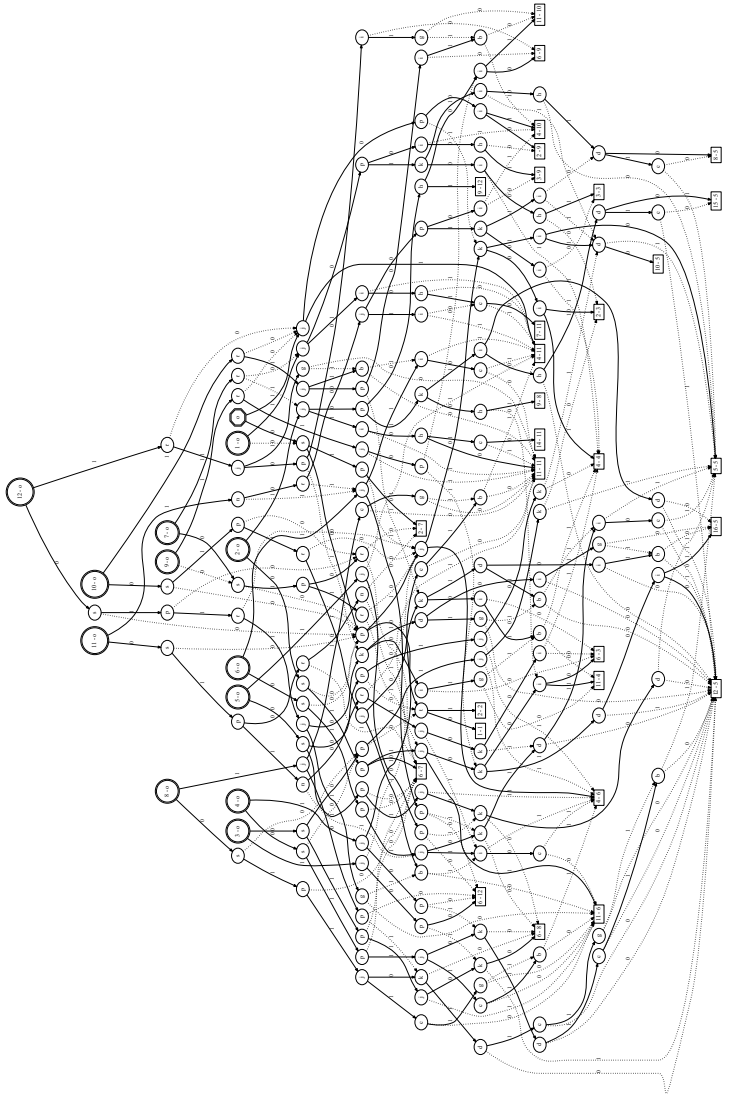


Figure 3.12: Forest of main trees connected into a single DAG with multiple entry points (roots). See Fig. 3.11 for details about notation used. Best viewed online.

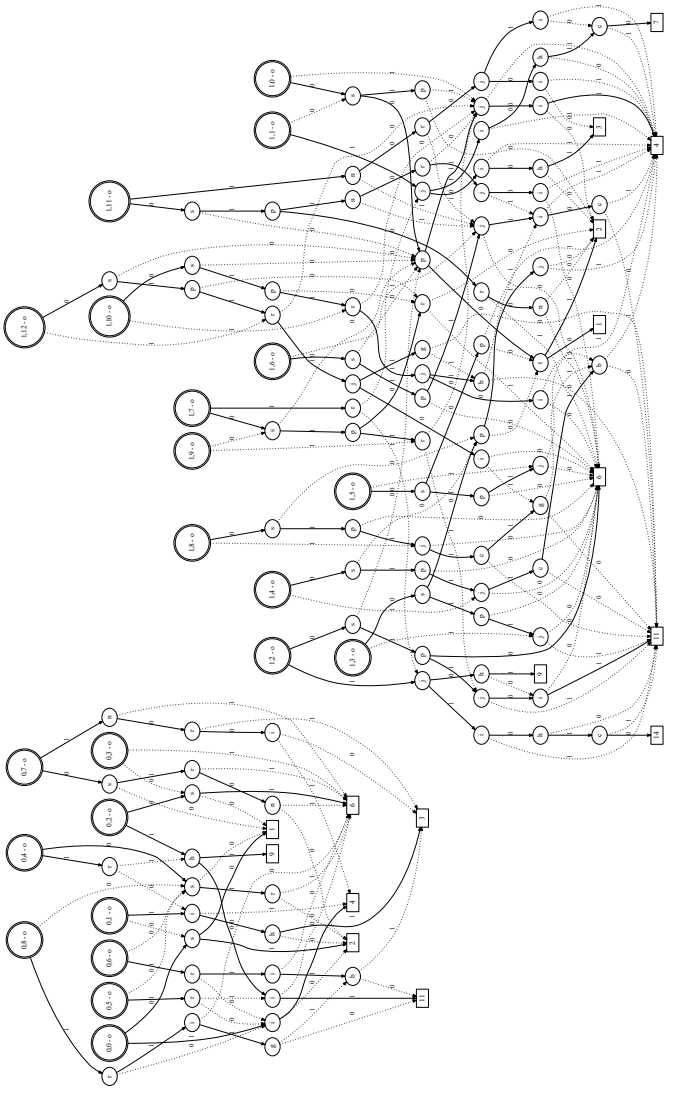


Figure 3.13: Forest of end trees connected into two groups of DAGs with multiple entry points (roots). See Fig. 3.11 for details about notation used. Best viewed online.

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

    if (CONDITION_0) {
        if (CONDITION_J) { // Leaf:
            ACTION_4       // do action and
            goto root_11;  // jump to next root
        }
        else {
...
root_6:
    c += 2; // Move to next block
    if (c >= cols - 2) {
        // Manage the last column/s
        goto last_column_6;
    }
    if (CONDITION_0) {
            NODE_80: // Name this node, because it will be reused
            if (CONDITION_J) {
...
root_11:
    c += 2; // Move to next block
    if (c >= cols - 2) {
            // Manage the last column/s
            goto last_column_11;
    }
    if (CONDITION_0) {
            if (CONDITION_N){
                    goto NODE_80; // Jump to existing subtree
            }
            else {
                    if (CONDITION_R){
...
```

Listing 3.1: *C++* like pseudo-code example of the Spaghetti algorithm. It is possible to observe the logic of block advancing (both on rows and columns), the action performed in leaves, the jump to the next tree, and the jump within conditions to reuse existing subtrees.

### 3.3.3 Implementation

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

The labeling process starts, at the beginning of each row, with the proper start tree, as shown in Fig. 3.10.

At the beginning of every main tree, the column index is incremented and an end of row check is performed, in order to decide whether an end tree should be used in place of the main tree. Anyway, two different end trees correspond to any main tree. This allows to cover both the case the image columns are in even number and the case they are odd. Therefore, if an end of row condition is really met, a check on the number of column is needed. Then, a *goto* statement causes a jump to the proper end tree. The two possible end trees are fixed for each main tree.

After the traversal of an end tree, the column index is reset, the row index is increased and an end of column check is performed. If the image is not over yet, a *goto* statement moves the execution flow to the beginning of the appropriate start tree, and the aforementioned process continues on the next row. The *C++* like pseudo-code provided in Listing 3.1 exemplifies all the process.

## 3.4 Benchmarking

The term *reproducible research* is usually referred to the approach of presenting scientific claims together with all information needed to reproduce the results, so that others may verify the findings and build upon them. As researchers, we strongly believe in and support reproducibility in scientific research. For this reason, in 2016 we publicly released the first version of YACCLAB [78] —Yet Another Connected Components LAbeling Benchmark—, an open source benchmarking system that allows to fairly compare CCL algorithms on different environments. YACCLAB has been originally described in [79] and later extended in [80] that included more literature algorithms and additional evaluation criteria, making it more general and flexible. In 2019, a new version of the benchmark including the possibility of evaluating 3D algorithms and GPU-based implementations has been released [62]. The benchmark represents nowadays the *de-facto* standard for evaluating CCL proposals, being it employed by several

authors [81, 81, 70, 82, 83].

When measuring the performance of an algorithm, many different subtleties must be taken into account. Although datasets have been blamed for narrowing the focus of research reducing it to a single benchmark performance number, it is clear that the ability to compare different techniques on the same data is essential to choose which algorithm suits specific needs best [84]. In order to ensure a common and standard test set, YACCLAB is provided with an open-source binary dataset, which covers most of the CCL application scenarios, including both 2D images and 3D volumes. Since the whole project is open source, anyone can contribute by increasing and improving the YACCLAB dataset with different application environments. The current set of images available in the benchmark and used to compare algorithm performance in this survey is described in Section 3.4.1. Before the publication of YACCLAB, many novel proposals have been published and almost none of them compared on the same data [69, 57] making performance comparison almost impossible.

Benchmarking may be a problem by itself, because measuring performance may not be obvious, but CCL has an exact result and this reduces the burden of the evaluator to the measurement of how fast an algorithm is. However, setting the correct parameters when reproducing other people tests may be a problem, changing the final figures by orders of magnitude. This is why publishing the source code together with a scientific paper or, at the very least, an executable should be mandatory. YACCLAB tackles these aspects by including the source code of all the most significant proposals of the last twenty years, considering both CPU- and GPU-based algorithms. All the implementations have been taken from the original publication, whenever available, or produced by following the paper pseudo-code and description. Being the code open source, anyone can check whether implementation details are correctly handled or not, and test algorithms on their own setting (operative system, compiler, dataset, etc.), verifying any claim found in the literature and selecting the algorithm that perform the best on their specific environment.

### 3.4.1 Evaluation Dataset

Following a common practice in literature [85, 37, 86], the YACCLAB dataset includes both synthetic and real images, addressing both 2D and 3D scenarios. All images are provided in 1 bit per pixel PNG format, with 0 being


Figure 3.14: Samples of the YACCLAB 2D datasets. Clockwise from the top-left: 3DPeS, Fingerprints, Medical, MIRflickr, Tobacco800, XDOCS, Hamlet.

background and 1 being foreground. The dataset can be automatically downloaded during the setup of the YACCLAB benchmark.

### 2D Datasets

*MIRflickr [87].* This is the Otsu-binarized [88] version of the MIRflickr dataset. It contains a set of 25 000 natural images taken from Flickr with few connected components and an average density of 0.4459 foreground pixels.

*Medical [89].* This dataset covers application of CCL on medical data. It is is composed of 343 binary histological images with an average amount of 1.21 million pixels to analyze and 484 components to label. This dataset is intended to cover applications of CCL related to medical image analysis.

*Hamlet.* A set of 104 images, scanned from the Hamlet provided by the Gutenberg Project [90]. Images have an average amount of 1 447 components to label and an average foreground density of 0.0789.

*Tobacco800.* Composed of 1 290 document images, it has been collected and scanned using a wide variety of equipment over time. Image sizes range from $1200 \times 1600$ to $2500 \times 3200$ pixels [91, 92, 93].

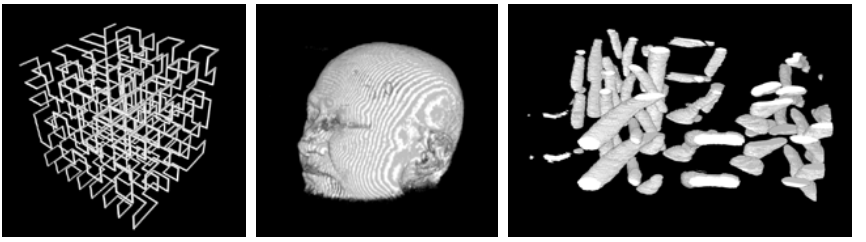


Figure 3.15: Samples of the YACCLAB 3D datasets. From left to right: Hilbert space-filling curve, OASIS and Mitochondria medical imaging data.

*XDOCS.* A collection of Italian civil registries recently digitalized [52, 94, 95]. This dataset is composed of 1 677 high resolution images with an average size of 4 853 × 3 387 and 15 282 components to analyze. The average foreground density is quite low: 0.0918.

*Fingerprints.* This dataset counts 960 fingerprint images taken from three fingerprint verification competitions (FCV2000, FCV2002 and FCV2004) [96]. Images were collected using low-cost optical sensors or synthetically generated, binarized using an adaptive threshold [97] and finally negated. Their size varies from 240 × 320 up to 640 × 480 pixels.

*3DPes.* This is a set of images coming from the surveillance dataset 3DPeS (3D People Surveillance Dataset) [98]. This has been designed mainly for people re-identification in multi-camera systems with non-overlapped fields of view, although it can be exploited for people detection, tracking, action analysis and trajectory analysis. The background models for all cameras are provided by the authors, so a very basic technique of motion segmentation has been applied to generate the foreground binary masks, i.e., background subtraction and Otsu thresholding [88]. The analysis of the foreground masks to remove small connected components and to match the nearest neighbors is a common application for CCL.

### 3D Datasets

*Hilbert.* The Hilbert curve is a fractal space-filling curve described by David Hilbert. This curve represents a challenging test case for the labeling algorithms, and it has already been used by other authors to evaluate the performance of CCL algorithms [86]. For these reasons we included in

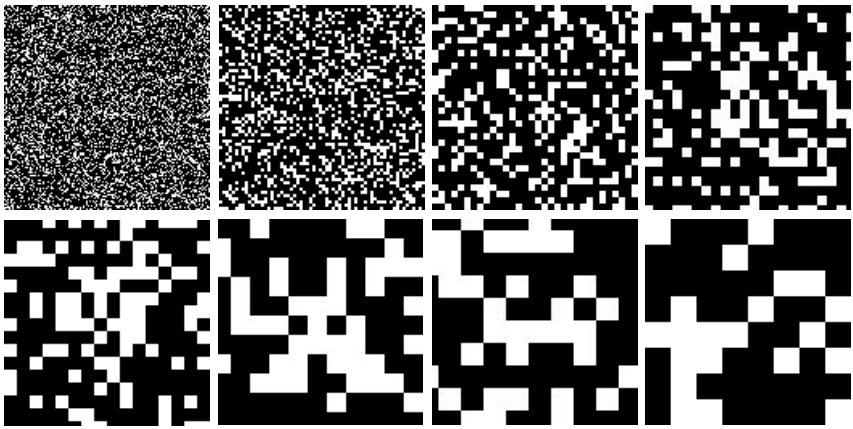


Figure 3.16: Samples of the YACCLAB 2D granularity dataset. Example images have a foreground density of 30% and, from left to right, top to bottom, granularities are 1,2,4,6,8,12,14,16.

the YACCLAB 3D-dataset six volumes filled with the 3D Hilbert curve obtained at different iterations (1 to 6) of the construction method. Final binary volumes have a size of $128 \times 128 \times 128$.

*OASIS.* It is a dataset of medical MRI data already used in the past to evaluate the performance of 3D CCL algorithms [86]. For this reason it was chosen and included in the YACCLAB dataset. All images in this test dataset were taken from the Open Access Series of Imaging Studies (OASIS) project [99]. The dataset consists of 373 volumes, each of which contains 128 slices of $256 \times 256$ pixels. To make the images suitable for CCL the original volumes were binarized with the Otsu threshold [88], calculated over the entire volume.

*Mitochondria.* This is the Electron Microscopy Dataset originally published in [100]. The original dataset contains sections taken from the CA1 hippocampus region of the brain. We included in YACCLAB all the binary volumes provided by the authors, for a total of three volumes composed by 165 slices with a resolution of $1024 \times 768$ pixels.

**Random Synthetic Images**

A set of black and white random noise images to stress how the behavior of algorithms varies with the *density* (percentage of foreground pixels) and *granularity* (minimum size of foreground blocks). Specifically, two different test sets are available in YACCLAB, one for 2D images and one for 3D volumes. The former contains $2048 \times 2048$ images, while the latter has $256 \times 256 \times 256$ volumes. Images and volumes have been generated with the Mersenne Twister MT19937 random number generator, included in the *C++* standard library [101]. Density ranges from 0% to 100% with a step of 1%. For every density value, each integer granularity $g \in [1, 16]$ has been considered. Ten images have been generated for every couple of density-granularity values, for a total of 16 160 images, both for 2D and 3D.

### 3.4.2 Assessment Strategies

The YACCLAB benchmark provides different kind of tests that can be used to stress algorithm behavior under different perspectives. The simplest evaluation mechanism consists in a direct comparison of execution times, including the time for allocating data structures. Algorithms are run on all the available images, and the average execution time per dataset is recorded.

The benchmark also allows to separately evaluate the different phases composing the algorithms, separating the memory allocation time from the core execution of the labeling procedure. Two-stage algorithms can also distinguish between the *First Scan* and the *Second Scan* times. The impact of each phase can be stressed and analyzed trough such a kind of experiments, highlighting strength and weaknesses of each proposal. *Granularity* is the last experiment available and consists of measuring the execution time on synthetic images when foreground pixels density and granularity change. This approach allows to draw additional conclusions, highlighting behaviors that may not emerge from the previous tests.

## 3.5 Comparative Evaluation

In this Section the benefits of Spaghetti Labeling are evaluated using YACCLAB. The fairness of the comparison is guaranteed by compiling the al-

gorithms with the same optimizations and by running them on the same data and over the same machine.

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

The algorithms provided by YACCLAB cover most of the paradigms for CCL explored in the past. We selected the most significant ones, *i.e.* the best performing according to [80], in order to showcase the performance of the proposed algorithm.

For convenience, the following acronyms will be used to identify algorithms: SAUF is the Scan Array-based Union-Find algorithm by Wu *et al.* [64], BBDT is the Block-Based with Decision Trees algorithm by Grana *et al.* [29], CTB is the Configuration-Transition-Based algorithm by He *et al.* [57], PRED is the Optimized Pixel Prediction by Grana *et al.* [74], DRAG represents the Direct Rooted Acyclic Graph algorithm introduced by Bolelli *et al.* [60] and described in Section 3.2.3. Moreover, NULL is a lower bound limit for all CCL algorithms over a specific dataset/image, obtained by reading once the input image and writing it on the output again. Finally, the proposed method is identified as *Spaghetti*.

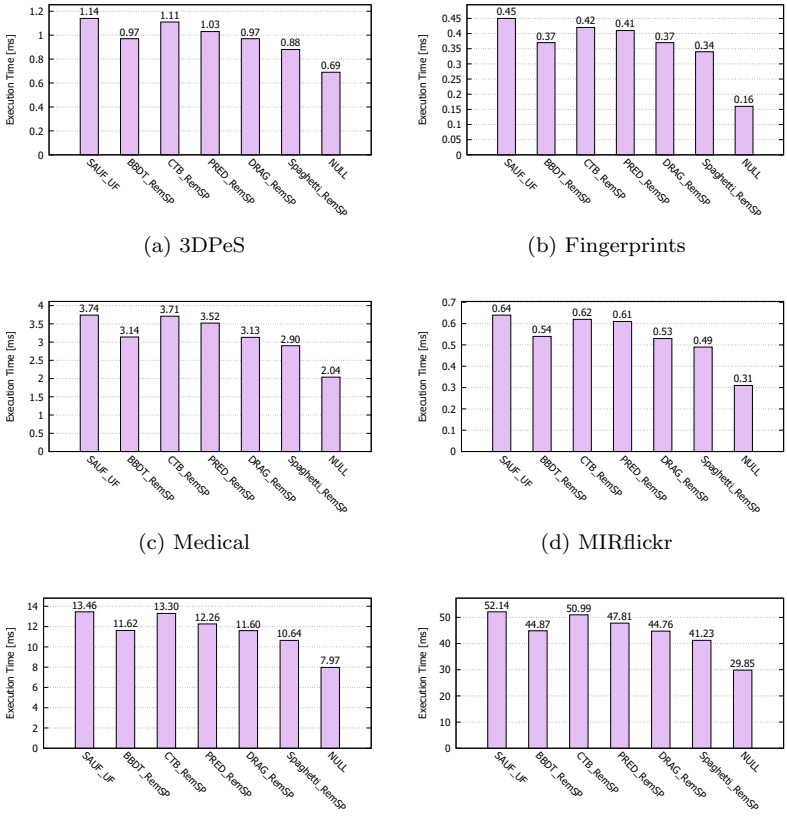
The benchmark provides a template implementation of the algorithms over the labels solving strategy. Using different label solvers can significantly change the performance of a specific combination of dataset, algorithm and operating system. To increase the readability of charts without losing information, we select, according to [80], the labels solvers that provide the best performance on the selected environment for a specific algorithm, *i.e.* standard *union-find* (UF) for SAUF and the interleaved Rem algorithm with SPlicing (RemSP) [41] for all the others.

### 3.5.1 Average Run-Time Results

For a given dataset, the set of algorithms is randomly shuffled, and each of them is run for a total of 10 iterations on each image. The minimum execution times are then considered and averaged on the dataset size [80]. Experimental results are reported in Fig. 3.17.

Algorithms that make use of Grana mask (BBDT, DRAG and Spa-

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




(a) 3DPeS

(b) Fingerprints

(c) Medical

(d) MIRflickr

(e) Tobacco800

(f) XDOCS

Figure 3.17: Average run-time tests in ms on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. Lower is better.

In order to highlight how the optimization introduced by our proposal influences the performance, a stacked bar chart obtained on the *Tobacco800* dataset is also reported in Fig. 3.18. In this experiment, the performance of an algorithm is evaluated splitting the allocation-deallocation (*alloc*/*dealloc*) time from the one required to compute CCL.

Moreover, each scan involved in the labeling procedure is displayed separately. It is important to underline that *alloc/dealloc* is an upper bound of the real allocation time [80], and this explains why execution times in Fig. 3.18 are higher than the ones in Fig. 3.17e. As it can be seen, the allocation time of SAUF, DRAG and Spaghetti is the same. Indeed, they require the same data structures to compute labeling, *i.e.* the output image and the vector to solve the labels equivalences. The NULL algorithm has a faster allocation step since it does not need an additional equivalences vector. During the *first scan*, temporary labels are assigned to the output image and possible equivalences between them are stored in the equivalence vector. During the *second scan* the provisional values are replaced with final labels. For what concerns these steps the following conclusions can be draw:

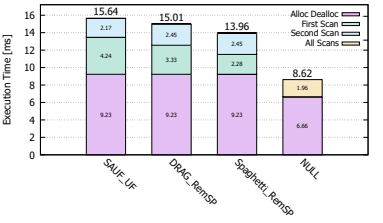



Figure 3.18: Average run-time tests with steps in ms on the *Tobacco800* dataset, obtained on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. Lower is better.

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

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

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

### 3.5.2 Load/Store Operations

In Table 3.1, the average number of load/store operations for each algorithm on the *Tobacco800* dataset is reported. The goal is to observe how the choice of the mask and state prediction affects read and write operations, and thus the performance of an algorithm. It is important to notice that counts displayed in the table do not distinguish between cache hits and misses. However, they allow to draw more detailed conclusions than raw execution times.

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

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

Table 3.1: Average number of load/store operations on the *Tobacco800* dataset. Quantities are given in millions.

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

SAUF and PRED over the equivalence vector/s is solely imputable to the different label solvers.

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

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

### 3.5.3 Synthetic Images

Following a common practice in literature [57, 29, 74, 80], we test the performance on images with varying density and size, taken from *density* and *granularity* datasets of the YACCLAB benchmark. Experimental results obtained on the former are reported in Fig. 3.19. It can be noticed that state-of-the-art CCL algorithms are linear in the number of pixels. The difference relies on the y-intercept of the lines. The gap observable around $10^5$ pixels is easily explainable taking into account that images do not fit L2 cache anymore, and require L3 cache to be employed.

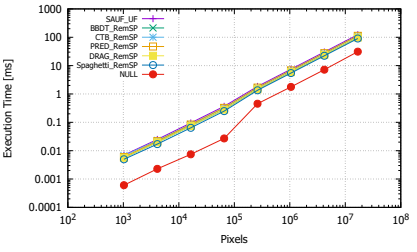
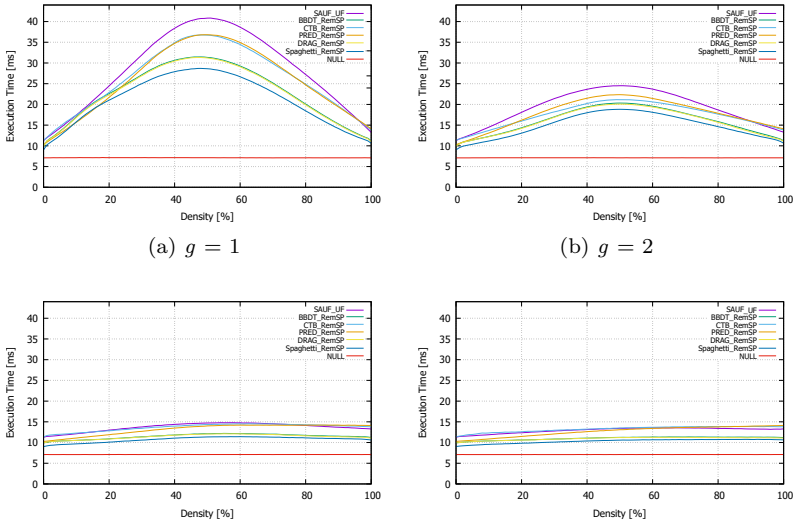



Figure 3.19: Experimental results in ms on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler obtained using images of increasing size. Lower is better.

On the other hand, results obtained with different values of granularity (1, 2, 8, and 16) are depicted in Fig. 3.20: the behaviour of all algorithms is strictly linked to the number of foreground pixels. More in detail, the parabolic trend can be explained considering the effects of the branch prediction unit, which heavily affects the algorithms around 50%. Focusing on *granularity*-1, it can be noticed that CTB and PRED algorithms have an analogous behaviour, since they

share a very similar paradigm, though realized differently. Benefits introduced by state prediction can be appreciated when comparing these two algorithms to classic SAUF. BBDT and DRAG also have similar behaviour to each other, with the second being slightly better thanks to the code compression technique. The Spaghetti algorithm, combining all previous improvements, shows the best performance at most densities.

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




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

(c) $g = 8$ (d) $g = 16$

Figure 3.20: Granularity results in ms on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. Lower is better.

## 3.6 Conclusion

This Section combined all the successful techniques that improved CCL algorithms performance, producing an extremely fast solution. The proposed approach showed superior performance, beating the results obtained by all compared approaches in all settings. In order to achieve this result an automatic simplified trees generation was required, along with a solution to leverage all the savings obtainable at image edges, *i.e.* the removal of *if* statements. Moreover, a greedy algorithm was designed allowing to convert decision forests into DAGs and thus reducing the machine code size more than a compiler could ever do. This is possible thanks to the concept of equivalent subtrees. The source code of the proposed strategy has been included in YACCLAB [78].

# Chapter 4

# One DAG to Rule Them All

> One Ring to rule them all, One Ring to find them, One Ring to bring them all, and in the darkness bind them
>
> J.R.R. Tolkien

This Chapter presents GRAPHGEN, a general open-source framework for optimizing the performance of many binary image processing algorithms. Starting from just a set of rules, it automatically generates decision trees with minimum path-length considering image pattern frequencies, then applies state prediction and code compression producing Directed Rooted Acyclic Graphs (DRAG) that combine these different optimization techniques. We showcase the framework on three classical and widely employed algorithms (namely Connected Components Labeling, Thinning and Contour Tracing). When compared to previous approaches —in 2D and 3D—, implementations using the generated optimal DRAG perform significantly better than previous state-of-the-art algorithms, on real-world and synthetic datasets.

Many of the optimizations included in the GRAPHGEN framework have already been introduced and explained in the previous Chapter about Spaghetti Labeling. Anyway, they are here summarized for easier reading of

this Chapter.

## 4.1 Introduction

In this Chapter, we introduce a novel framework that allows to automatically apply the most effective optimization strategies presented in literature to any problem modelled with Decision Tables (DT). The framework, called GRAPHGEN (the all encompassing GRAPH GENerator), takes a definition of the problem as input, in terms of conditions that need to be checked and actions that have to be performed, and produces *C++* code, which implements the desired solution with all required optimizations as output.

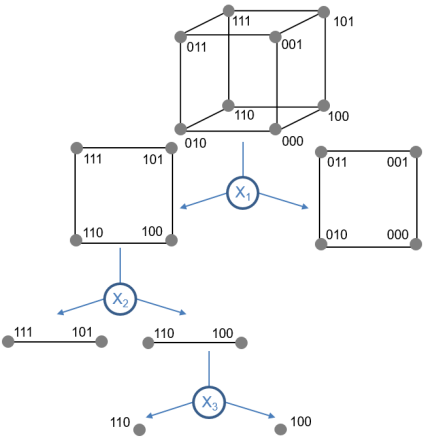



Figure 4.1: Example of 3-dimensional hypercube partitioning.

To demonstrate the capabilities of GRAPHGEN, we selected three well-known 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 algorithmic solutions available in literature, apply these strategies to different problems, and even combine them to enhance performance and significantly improve state-of-the-art algorithms. To prove the generality of GRAPHGEN, the presented benchmarks are not limited to 2D sequential image processing algorithms, but also include 3D scenarios as presented in Section 4.5. Our main contributions can be summarized as follows:

1. GRAPHGEN, a generalized decision tree/forest generation framework that can apply various optimizations such as state prediction, compression and optimization based on image frequencies automatically,

on all problems that are based on decision tables such as CCL, Thinning and CC.

2. An improvement to the compression strategy of trees/forests of Spaghetti [102] to be optimal, therefore further optimizing the performance through smaller trees.

3. The best-performing generated implementations of GRAPHGEN in 2D CCL, 3D CCL, Thinning and Chain Code are significantly faster than recent state-of-the-art algorithms and OpenCV implementations (section 4.5).

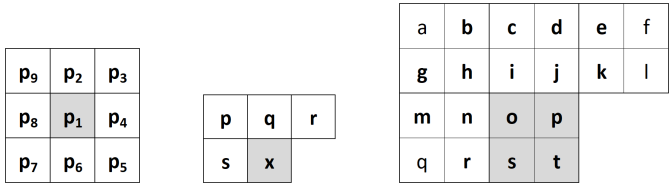
The source code of the framework, and the complete documentation are available in [103].

## 4.2 Modelling Algorithms with Decision Tables

In the analysis of binary images most algorithms share the same general structure: for each (group of) pixel $x$ the action to be performed depends on the value of $x$ and its neighborhood. The set of pixels whose value can condition the action constitutes the *mask*. As seen in Chapter 3, multiple possible masks can be defined; the most common ones are depicted also in Fig. 4.2, for reference. 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 way in which the neighborhood is explored. This applies to 2D and 3D images but also when addressing parallel CPU- and GPU-based environments.

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 complexity of these algorithms.

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 provides a unified strategy, able to automatically apply and combine the most effective solutions to access, at each




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

Figure 4.2: 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 $3 \times 3$ neighborhood in 8-way connectivity, (b) and (c) are improved versions commonly employed in CCL.

step of an algorithm, only the pixels which are effectively required, while reducing the corresponding machine code.

The class of algorithms that GRAPHGEN deals with can be described with a *command execution metaphor* [29]. This concept has already been introduced in the previous Chapter, but a formal mathematical definition is given in this Section.

The values of pixels in the mask constitute a rule (binary string), to which a set of equivalent actions is associated. 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. 4.3). The *OR*-decision table can be easily translated into a LookUp 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 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 $a$, if $p$ and $x$ have value 1 the action to be performed is already known, and there is no need to check $q$. Consequently, some processing time can be saved removing unnecessary pixel checks, by making use of 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 leaves contain actions, 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 [76] with a dynamic programming approach and extended in [72] to *OR*-decision tables.

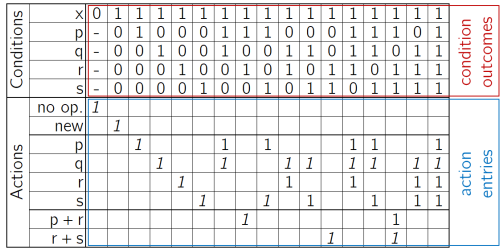



Figure 4.3: *OR*-decision table for Rosenfeld mask adopted in CCL.

The conversion of a decision table (with $L$ conditions) to a DTree can be interpreted as the partitioning of an $L$-dimensional hypercube (Fig. 4.1) ($L$-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 4.2.0.1** (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 $\ell$ corresponds to a $k$-cube, denoted by $K_\ell \subseteq K$, and the set of $K_\ell$ is a partition of $K$;
2. each leaf $\ell$ has a non empty set of actions $A_{K_\ell}$, associated to cube $K_\ell$ 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.* $A_K \neq \emptyset$).

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:

$$gain(T) = \sum_{\ell \in \mathcal{L}} \left( P_{K_\ell} \sum_{i \in D_{K_\ell}} w_i \right) \tag{4.1}$$

A tree with maximum gain for a $k$-cube is called Optimal Decision Tree (ODT), and can be built by means of a recursive procedure. At step 0, all 0-cubes are associated to a gain of 0 according to the formula. At step $n$, the algorithm builds all the 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 $S$:

$$Gain_S = \max_{i \in D_S} (Gain_{S_{i,0}} + Gain_{S_{i,1}} + \delta w_i P_S) \tag{4.2}$$

Where $\delta$ 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. Grana *et al.* proved the correctness of the algorithm in [72].

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.

```yaml
pixel_set:
    pixels:
        - {name: p, coords: [-1, -1]}
        - {name: x, coords: [ 0, 0]}
        - ...
    shifts: [1, 1]
conditions: [p, q, r, s, x]
actions: [noop, x<-new, x<-p, ..., x<-r+s]
rules: [[1], [1], ..., [3, 4, 5, 7]]
```

Listing 4.1: Example of YAML configuration file which defines the SAUF CCL algorithm [64] 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*).

The generation of an optimal decision tree is the first step of any optimization process provided by the framework. Given a YAML file that defines a task in terms of conditions and actions (Listing 4.1), GRAPHGEN is able to generate the optimal decision tree and the associated source code.

## 4.3 State Prediction

The second optimization step of GRAPHGEN, *prediction*, concerns the exploitation of the information gathered in the previous step that can also be useful for the current one. As explained in Chapter 3, prediction can be used whenever some pixels are still part of the mask after the shift.

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. 4.4, pixels from $p_1$ to $p_6$ may be used again after moving to the next pixel, 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 [104]. He *et al.* [57] 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.

In [74], the authors 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. It is noteworthy that, for a certain position of the mask, the set of pixels that theoretically do not need to be read are not only the already seen ones, but also pixels that are outside the input image, which are usually considered background. In fact, Grana *et al.*also exploited the information about the position of the mask in the image, and built special reduced DTrees that are only used in correspondence with borders. The whole optimization strategy has been applied *by hand* to the SAUF algorithms, *i.e.* using the Rosenfeld mask.

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.

### 4.3.1 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 constraints set is filled for each leaf of

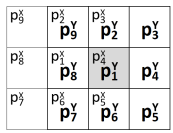



Figure 4.4: Unitary horizontal shift for the $3 \times 3$ mask during image scan. Pixels named with "X" were inside the mask in the previous iteration while pixels named with "Y" are currently inside.

the general DTree, and is used to create a reduced version of it, that is meant to be used to determine the action for the next pixel instead of the general one. 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, that 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.

### 4.3.2 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, that 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, these boundary checks can be deleted.

## 4.4 From Trees to DRAGs

The ODT generated in the first step of GRAPHGEN optimization procedure (Sec. 4.2) can contain identical or equivalent subtrees, as we noticed in [60]. We noticed how those subtrees could be merged together, reducing the size of the compiled machine code. The formal statement of the problem is provided in Section 3.2.3.

As said, 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 anticip-

ated 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. In GRAPHGEN, compression can be directly applied to an ODT or to a decision forest obtained through state prediction. It is also possible to merge together subtrees of different trees, as long as they belong to the same forest. Conversely, subtrees of different forests must stay separated, to preserve the jumps between trees.

The compression of a forest into a multi-rooted DRAG is performed, in GRAPHGEN, 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 [77]. 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.

The strategy used in GRAPHGEN is an exhaustive recursive procedure that tries every possible 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 remain. 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 (the greedy strategy just described is the one we used for the generation of Spaghetti Labeling). This holds true in our experience: in all the examples that we tried, the best solution found by the exhaustive algorithm is the first one.

## 4.5 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 YACCLAB dataset, introduced in Section 3.4.

### 4.5.1 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 described in detail in the previous Chapter, so we will skip directly to experimental results.

For comparing the GRAPHGEN generated algorithms with existing ones, the open-source benchmarking system YACCLAB [80, 79] 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.

Starting from the Rosenfeld mask, generating an ODT recreates SAUF, the reference algorithm for the following Average Speed-Up (ASU) comparisons (Table 4.1). Then, applying state prediction provides the PRED algorithm (ASU=1.103), and finally, trying to compress the forest into a DRAG, we obtain the same ASU with PRED++: since the PRED forest is already very small, reducing the code size does not affect the usage of the instruction cache.

Tackling the problem with the Grana mask already proved to be an effective idea, and its GRAPHGEN-generated ODT 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 pre-

serving 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 4.2, GRAPHGEN is able to consider image frequencies when generating the ODT. 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).

Table 4.1: Average run-time experimental results on 2D CCL algorithms in milliseconds. ASU is the Average Speed-Up over SAUF. The star identifies novel variations on previous algorithms generated thanks to GRAPHGEN. Lower is better.

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

It is important to notice that, to be as fair as possible, frequencies are calculated over the entire YACCLAB dataset, but it is easy to specialize them for specific problems.

### 4.5.2 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 [105].

Many thinning algorithms belong to the class of parallel thinning algorithms [106]: every pixel is analyzed considering its neighborhood values in the current image, but the result is written into a different output mask,

so that the procedure can be easily implemented on massively parallel architectures.

We consider three classical algorithms: *(i)* the algorithm proposed by Zhang and Suen (ZS) in [107] 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, *(ii)* Chen and Hsu (CH) fixed some corner cases, and proposed a LookUp Table (LUT) to speed-up the process [108]. Furthermore, *(iii)* Guo and Hall [109] proposed a solution to better cope with 2×2 squares and diagonal lines, obtaining skeletons with less stair case artifacts. These solutions have been proposed some decades ago, but are still commonly used [24, 25, 26] and included in many image processing libraries, such as OpenCV.

All three algorithms follow the same base structure: at each iteration both subiterations must be performed and if neither of them modifies the image, the algorithm finishes. At each subiteration, the image is scanned and for each foreground pixel we check if the pixel should be removed (Algorithm 2).

The conditions for foreground pixel removal depend on the neighborhood and the algorithm flavor. Following the original notation of Zhang and Suen, pixels are enumerated in clockwise order, with the current pixel being $P_1$.

**Algorithm 2** Two subiteration thinning algorithm

```
1:  function ITERATION(I, O, k)
2:      O ← I
3:      changed ← false
4:      for all p ∈ L(I) do
5:          if I(p) = 1 then
6:              if SHOULD_REMOVE(I, p, k) then
7:                  O(p) ← 0
8:                  changed ← true
9:      return changed

10: procedure THINNING(I, O)
11:     repeat
12:         changed_0 ← ITERATION(I, O, 0)
13:         changed_1 ← ITERATION(O, I, 1)
14:     until ¬changed_0 ∧ ¬changed_1
```

Algorithm 3 and Algorithm 4 provide a detailed summary of the algorithms proposed by Zhang and Suen, Chen and Hsu and Guo and Hall. In all of them, $k$ represents the subiteration index: $k = 0$ during the first subiteration and $k = 1$ during the second one. Support logic functions are used such as $A(P)$, which is the number of 01 patterns in clockwise order, and $B(P)$, which is the number of non zero neighbors of $P_1$.

**Algorithm 3** Removal logic functions for ZhangSuen and ChenHsu algorithms

1: **function** A($P$)
2: **return** $(\neg P_2 \wedge P_3) + (\neg P_3 \wedge P_4) + (\neg P_4 \wedge P_5) + (\neg P_5 \wedge P_6) +$
3: $(\neg P_6 \wedge P_7) + (\neg P_7 \wedge P_8) + (\neg P_8 \wedge P_9) + (\neg P_9 \wedge P_2)$
4: **function** B($P$)
5: **return** $P_2 + P_3 + P_4 + P_5 + P_6 + P_7 + P_8 + P_9$

6: **function** ZS_SHOULD_REMOVE($I$, $p$, $k$)
7: $P \leftarrow I(\mathcal{N}(p))$
8: **if** $k = 0$ **then**
9: $c \leftarrow P_2 \wedge P_4 \wedge P_6$;
10: $d \leftarrow P_4 \wedge P_6 \wedge P_8$;
11: **else**
12: $c \leftarrow P_2 \wedge P_4 \wedge P_8$;
13: $d \leftarrow P_2 \wedge P_6 \wedge P_8$;
14: **return** $(\mathrm{A}(P) = 1) \wedge (2 \leq \mathrm{B}(P) \leq 6) \wedge \neg c \wedge \neg d$

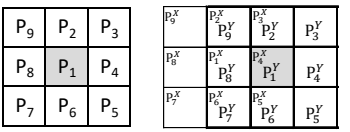
15: **function** CH_SHOULD_REMOVE($I$, $p$, $k$)
16: $P \leftarrow I(\mathcal{N}(p))$
17: **if** $k = 0$ **then**
18: $c \leftarrow P_2 \wedge P_4 \wedge P_6$;
19: $d \leftarrow P_4 \wedge P_6 \wedge P_8$;
20: $f \leftarrow P_2 \wedge P_4 \wedge \neg P_6 \neg P_7 \neg P_8$
21: $g \leftarrow P_4 \wedge P_6 \wedge \neg P_2 \neg P_8 \neg P_9$
22: **else**
23: $c \leftarrow P_2 \wedge P_4 \wedge P_8$;
24: $d \leftarrow P_2 \wedge P_6 \wedge P_8$;
25: $f \leftarrow P_2 \wedge P_8 \wedge \neg P_4 \neg P_5 \neg P_6$
26: $g \leftarrow P_6 \wedge P_8 \wedge \neg P_2 \neg P_3 \neg P_4$
27: **return** $(2 \leq \mathrm{B}(P) \leq 7) \wedge ((\mathrm{A}(P) = 1) \wedge \neg c \wedge \neg d) \vee$
28: $(\mathrm{A}(P) = 2) \wedge (f \vee g))$

**Algorithm 4** Removal logic function for GuoHall algorithm

```
1: function GH_SHOULD_REMOVE(I, p, k)
2:     P ← I(N(p))
3:     C ← ((¬P2) ∧ (P3 ∨ P4)) + ((¬P4) ∧ (P5 ∨ P6))+
4:           ((¬P6) ∧ (P7 ∨ P8)) + ((¬P8) ∧ (P9 ∨ P2))
5:     N1 ← (P9 ∨ P2) + (P3 ∨ P4) + (P5 ∨ P6) + (P7 ∨ P8)
6:     N2 ← (P2 ∨ P3) + (P4 ∨ P5) + (P6 ∨ P7) + (P8 ∨ P9)
7:     N ← min(N1, N2)
8:     if k = 0 then
9:         m ← (P6 ∨ P7 ∨ ¬P9) ∧ P8
10:    else
11:        m ← (P2 ∨ P3 ∨ ¬P5) ∧ P4
12:    return (C = 1) ∧ (2 ≤ N ≤ 3) ∧ ¬m
```

The basic idea is to remove pixels at foreground connected components edges (*i.e.*, the block should not be totally foreground), without splitting that component.

Chen and Hsu [108] observed that given the eight neighbors of $P_1$ the outcome of the conditions is known, thus they built two LUTs for the two subiterations and used the pixel values as bits for the index of the LUT. This allows to save all the operations required to compute $A(P)$, $B(P)$ and the other two conditions, adding only one memory access. The same approach can obviously be applied to the GuoHall rules.




(a) (b)

Figure 4.5: Naming convention for the pixel in the neighborhood of $P_1$ (a) and their overlap when the mask is shifted for processing the next pixel (b).

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. 4.5a) have been optimized with GRAPHGEN and compared with the open-source framework THeBE (THinning evaluation

BEnchmark) [110]. THeBE is an extension of YACCLAB that we proposed in [111] to cope with skeletonization algorithms.

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 other have been generated using GRAPHGEN: the Tree version only employs an ODT, while the Spaghetti variants include state prediction and forest compression with DRAGs. The GRAPHGEN-generated DRAG for the Zhang and Suen algorithm is depicted in Fig. 4.6.

When comparing the various implementations within each algorithm (Table 4.2), 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.

Table 4.2: Average run-time experimental results of Thinning algorithms in milliseconds. The star identifies novel variations on previous algorithms generated thanks to 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** |

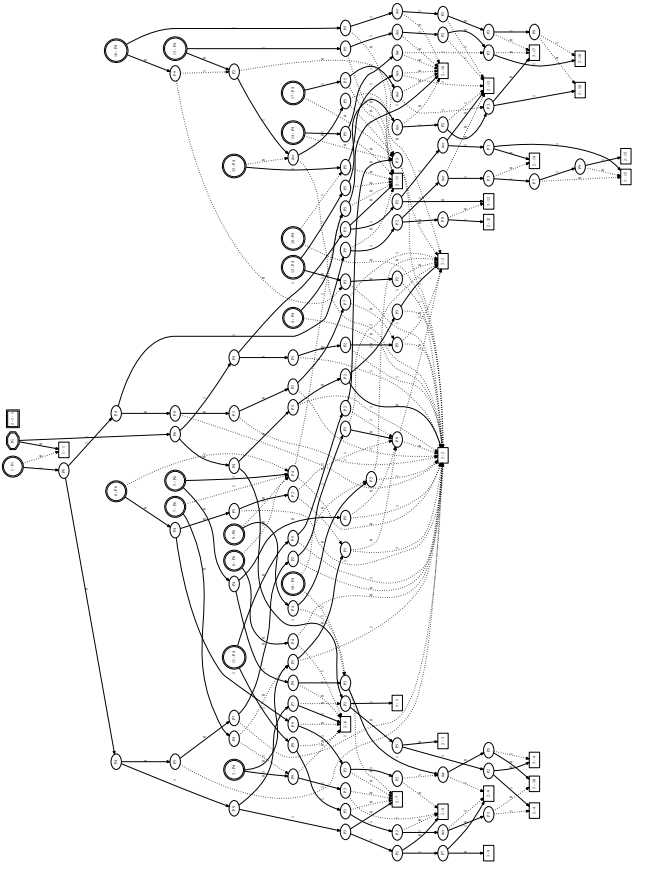


Figure 4.6: GRAPHGEN-generated DRAG for Zhang and Suen algorithm. The notation used is the same described in Chapter 3: the octagonal shaped root is the starting node for the first pixel in a line, double circles are roots from which the algorithm will restart after reaching a leaf (the first number is their id), ellipses are decisions and rectangles are leaves. The first number in leaves is the action to be performed (1=do nothing, 2=don't remove, 3=remove), while the second number is the next tree to be used. The special case (root 2) is marked as a double rectangle to stress that this root is also a leaf. Best viewed online.

### 4.5.3 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 [112]. 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. Since each pixel has only eight neighbors, it is sufficient to use a number from 0 to 7 to identify the next contour point.

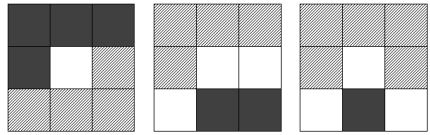
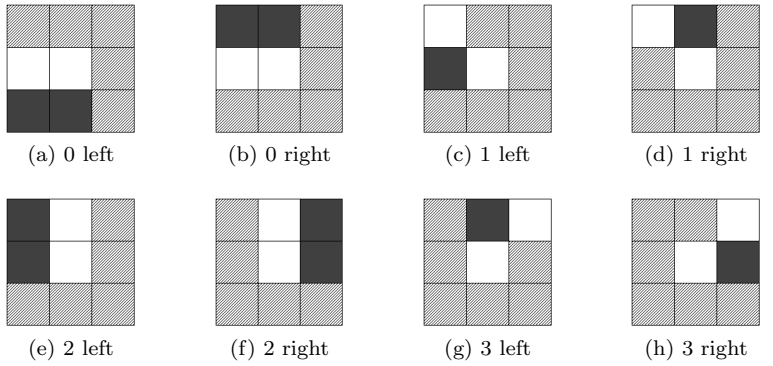



Figure 4.7: Max point templates. Dark squares are background, white squares are foreground and squares pattered with diagonal lines represent the concept of indifference. All outer max points correspond to template (a), while inner max points can either match (b) or (c).

Cederberg [113] proposed another variation, called Raster-scan Chain-code (RC-code), that could ease the retrieval when examining the image in a raster scan fashion.

In the RC-code, several coordinates are listed for each contour, and represent the first pixels that are hit in some 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. 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. A max point can either be an outer one, when it is a transition from background to object, or an inner one, when at object-background transition. They can be identified by templates in Fig. 4.7. When proceeding in raster scan, only four directions are possible, so a link is represented by a number from 0 to 3. Templates corresponding to chain links are depicted in Fig. 4.8. 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.



(a) 0 left (b) 0 right (c) 1 left (d) 1 right

(e) 2 left (f) 2 right (g) 3 left (h) 3 right

Figure 4.8: Chain link templates. Dark squares are background, white squares are foreground and squares patterned with diagonal lines represent the concept of indifference.

The merging of two R-chains consists in recognizing that the left R-chain continues the same contour traced by the right one, and therefore an ordering between max points of the same contour can be established. Min points templates are the same of max points ones, but upside-down.

An example of scan is depicted in Fig. 4.9. In Fig. 4.9a, a contour pixel is met for the first time, and is recognized as a max point ($M_0$). The first foreground pixel of the next row is part of the same contour, but in the moment that it is met by the raster scan it does not appear linked to any previously known chain, and so is labeled as max point as well ($M_1$). Then, in Fig. 4.9c, the scan reaches a min point that links together $M_0$ left chain and $M_1$ right chain. Fig. 4.9d shows the final result, where $M_2$ is an inner max point. An RC-code is 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. The RC-code can be converted to Freeman chain code following the same procedure. When computing the RC-code, is it sufficient to look at the 3×3 neighborhood of a pixel to recognize its nature, *i.e.*, whether it is a max point, a min point or a chain link.

In our implementation, the RC-code is an array of max points. A max point is represented by a *C++* structure detailed in Listings 4.2.

```
struct MaxPoint {
  unsigned row, col;
  Chain left, right;
  unsigned next;
}
```

Listing 4.2: Definition of the max point structure in *C++*.

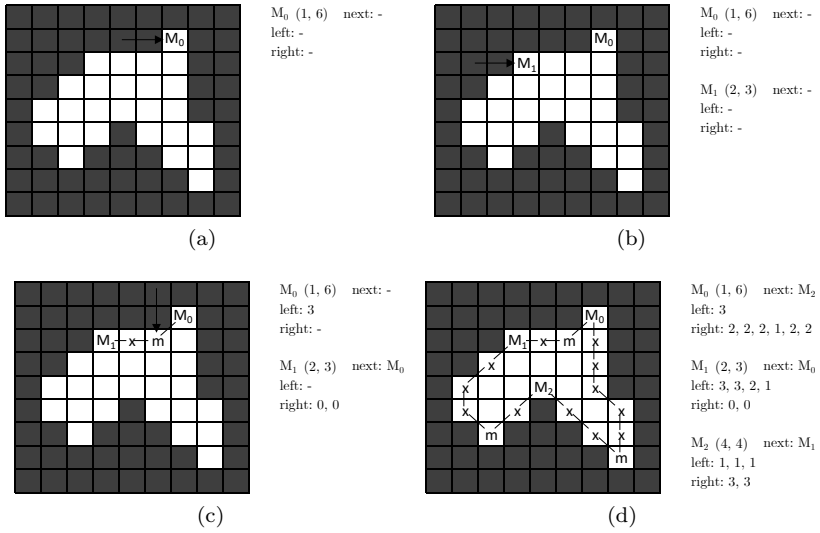



Figure 4.9: Example of execution of the RC-code extraction. The algorithm proceeds in raster scan, and in (a), (b) and (c) the arrow points to the pixel currently analyzed. Symbols M, m and x respectively represent max points, min points and chain links. To the left, max points are listed, along with their description.

Members *row* and *col* are the max point coordinates, *left* and *right* are the chains and *next* is the array index of the following max point. Multiple possibilities are viable for the implementation of the *Chain* class, the simplest of which is just the *vector* from the standard library. Anyway, since only 4 kinds of link exist, our *Chain* class has a more efficient

implementation that represents a link with only 2 bits.

The RC-code retrieval algorithm uses two data structures: a list of max points and their R-chains (*maxpoint_list*), and a list of active R-chains (*chain_list*), sorted in the order that the raster scan is supposed to meet them along the row. During the row scan, the variable *chain_pos* holds the position in *chain_list* of the next R-chain that is expected to be met. In this list, right and left R-chains alternate. To sum up, a pixel can be none, one or more of the following: outer/inner max or min point, and left/right link of type 0, 1, 2 or 3.

Thus, from the RC-code point of view, 12 boolean predicates (2 Max Points, 2 Min Points, 8 Links) totally describe a pixel. We say that they constitute the pixel *status*. For each pixel, the procedure is:

1. Retrieve the pixel status from its neighborhood.
2. Perform an action depending on the status.

The action is schematized in the following sequence of steps:

1. Move *chain_pos* backward for every link of type 0. Those are horizontal links, and must be attached to the previously met chain on the same row.
2. Iterate through links, from 0 to 3, left before right. For each one, attach it to the chain at position *chain_pos*, and increment *chain_pos*.
3. If inner min point, merge the last left R-chain met and the chain preceding it, then remove both from *chain_list*.
4. If outer min point, merge the last right R-chain met and the chain preceding it, then remove both from *chain_list*.
5. If inner max point, add a new element to *maxpoint_list*, and add its R-chains to *chain_list*, the right before the left one. The two chains must be inserted before *chain_pos*, or before *chain_pos - 1* if the last chain met is a right one.
6. If outer max point, add a new element to *maxpoint_list*, and add its R-chains to *chain_list* before *chain_pos*, the left before the right one.

In our implementation, the action is executed by a *C++* function composed of a sequence of *if* statements that takes the status as a parameter.

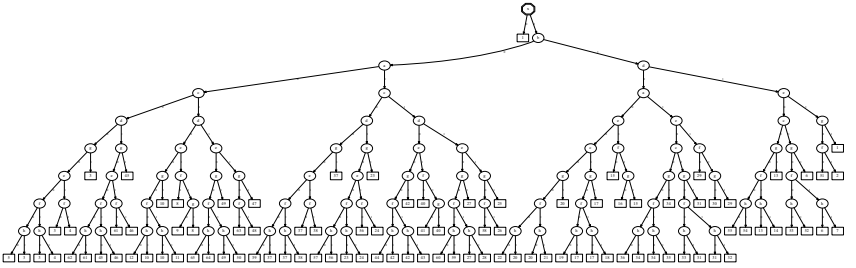


Figure 4.10: Binary decision tree which provides the Chain Code pixel status with minimal load/store operations. Best viewed online.

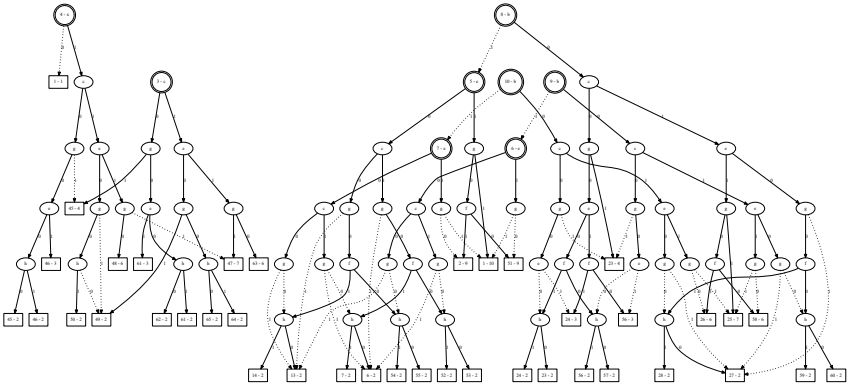
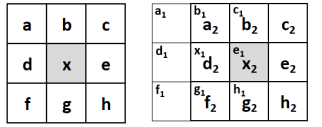


Figure 4.11: Part of the binary DRAG which provides the Chain Code pixel status with minimal load/store operations. Best viewed online.

When computing the RC-code, is it sufficient to look at the mask of Fig. 4.12 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. Thus, this algorithm is a suitable use case for GRAPHGEN.

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. The ODT and the DRAG obtained with our framework are depicted respectively in Fig. 4.10 and Fig. 4.11.




(a) (b)

Figure 4.12: (a) is the neighborhood mask, containing pixels whose value affects the status of $x$. (b) illustrates the concept of pixel prediction: 6 pixels of the mask are still inside the mask after the horizontal shift, though with different names.

In order to characterize the contour tracing performance, we adopted YACCLAB into BACCA (Benchmark Another Chain Code Algorithm). The source code is available in [114]. The reference algorithm is the one implemented by OpenCV 3.4.7 [115], which uses an extremely optimized contour following approach, while the algorithm proposed by Cederberg [113] has been implemented in multiple variants: using lookup tables with and without state prediction (LUT_PRED and LUT),

Table 4.3: Average run-time experimental results on chain code algorithms in milliseconds. ASU is the Average Speed-Up over OpenCV. The star identifies novel variations on previous algorithms generated thanks to GRAPHGEN. Lower is better.

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

a version based on optimal decision trees and another one again with state

prediction and compression (Spaghetti).

As it can be observed in the experiments (Table 4.3), LUT implementations fail to compete with the proven and carefully designed algorithm in OpenCV (ASU $< 1$). The GRAPHGEN-generated ODT already provides a significant speed-up (ASU=1.23), which is further improved by the Spaghetti versions (ASU=1.28).

### 4.5.4 What About 3D?

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. 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 to consider would be too high to compute within a reasonable timeframe with current computing capabilities. For instance, a complete, naive version of the BBDT mask in 3D would contain 112 voxels in the mask (14 blocks with 8 voxels per block) which in turn would require consideration of $2^{112} = 5.19 * 10^{33}$ rules. With 1 byte per rule, keeping a rule table in memory would require approximately $10^{24.68} TB$.

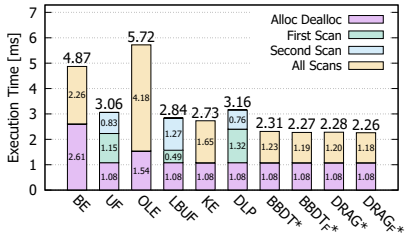


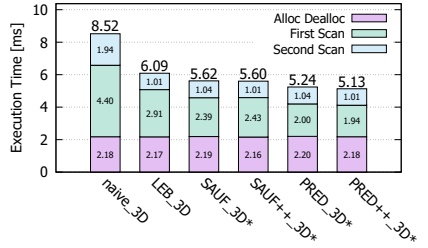

Figure 4.13: Average run-time experimental results on 2D CCL algorithms on GPU on the *Hamlet* dataset in milliseconds. The star identifies novel variations on previous algorithms generated thanks to GRAPHGEN. Lower is better.

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.

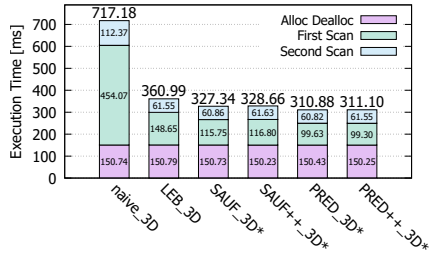
The algorithms that have been generated with GRAPHGEN for 3D CCL are SAUF_3D, using the optimal decision tree, and its DRAG-compressed version SAUF++_3D, and PRED_3D, which adds state prediction, and its compressed version PRED++_3D. As a comparison, a naive 3D

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.* [116] (LEB) were benchmarked. LEB is a heuristic that allows to construct a decision tree for the Rosenfeld 3D mask by hand, starting from the pixels with the most neighbors. Fig. 4.14 shows the effectiveness of our proposal also in this case.

PRED++_3D almost always performs the best due to the advantage of both utilizing state prediction and compression. However, on the Mitochondria dataset, compression seems to not increase the performance but actually worsen it in comparison to PRED_3D, similarly to PRED and PRED++ in 2D CCL. However, the difference is very small and PRED++_3D still performs better than every other approach by a significant margin, 16% faster than LEB_3D and 131% faster than the naive 3D CCL implementation.




(a) Hilbert




(b) Mitochondria

Figure 4.14: Average run-time experimental results on 3D CCL algorithms in milliseconds. The star identifies novel variations on previous algorithms generated thanks to GRAPHGEN. Lower is better.

## 4.6 Beyond the ODT: a Heuristic-Based Decision Tree for efficient CCL of 3D Volumes

This last Section presents a novel heuristic algorithm for Connected Components Labeling of 3D Volumes, based on decision tree learning methodology, called Entropy Partitioning Decision Tree (EPDT). It allows to compute near-optimal decision trees for large scan masks. Experimental results demonstrate that algorithms based on the generated decision trees

outperform state-of-the-art competitors.

As already seen in Section 4.5.4, extending a 2D algorithm to 3D is just a matter of increasing the size of the scanning mask to cover also the z-direction. Unfortunately, the generation of optimal decision trees becomes quickly unfeasible when the size of the scanning mask increases. In order to compensate the limitation of the optimal decision tree generation, this Section proposes a novel heuristic algorithm based on decision tree learning and named Entropy Partitioning Decision Tree (EPDT). The proposed solution allows to compute near-optimal decision trees for large scan masks, which was impossible with previous methods. The effectiveness of the proposed solution is demonstrated by applying the 2D block-based approach to a 3D scenario, and generating the corresponding decision tree.

### 4.6.1 Decision Tree Learning

Let us consider a population of samples, each composed of an array of discrete features, and each belonging to a certain class. A decision (or classification) tree is a tree where each node is labeled with a feature, and has as many children nodes as the different values for that feature. Finally, leaves are labeled with class names. Such a tree serves to determine the class of any sample from the population: this can be done by traversing the tree from root to leaf, choosing at each node the edge corresponding to the actual feature value of the sample.

The construction of a classification tree starts from an available labeled dataset, and usually proceeds top-down, choosing for the root node the feature that best splits the dataset according to a certain metric [117]. The process is repeated recursively on every node, considering only the subset of the dataset that reached it, and stops when every sample of the subset belongs to the same class. A common metric for evaluating splits is the Information Gain (IG), employed in the ID3 tree generation algorithm [118], and in its extension C4.5 [119]. The information gain represents how much uncertainty can be reduced by splitting on a certain feature. The uncertainty of a set $S$ is measured by its entropy $H(S)$, defined as:

$$H(S) = \sum_{c \in C} -p(c) \log_2 p(c) \tag{4.3}$$

In the formula, $C$ is the set of classes in $S$, and $p(c)$ is the inferred

probability of class $c$, calculated as the number of samples in class $c$ over the total number of elements in $S$. Information Gain $IG(S, f)$ is defined as the difference between the entropy of $S$ and the weighted average entropy of its subsets, determined splitting on feature $f$.

$$IG(S, f) = H(S) - \sum_{t \in T} p(t)H(t) \tag{4.4}$$

Where $T$ is the partition of $S$ produced by the split, and $p(t)$ is the proportion of the cardinality of $t$ to that of $S$. For each node, the feature yielding the largest information gain is chosen for the split. This process is a greedy heuristic, which performs a best-first search for locally optimal entropy values. In our specific case, each split will divide the original set $S$ into two subsets $T_0$ and $T_1$, with $p(T_0) = p(T_1) = 0.5$:

$$IG(S, f) = H(S) - 0.5 \cdot H(T_0) - 0.5 \cdot H(T_1) \tag{4.5}$$

### 4.6.2 Outline of the Proposed Algorithm

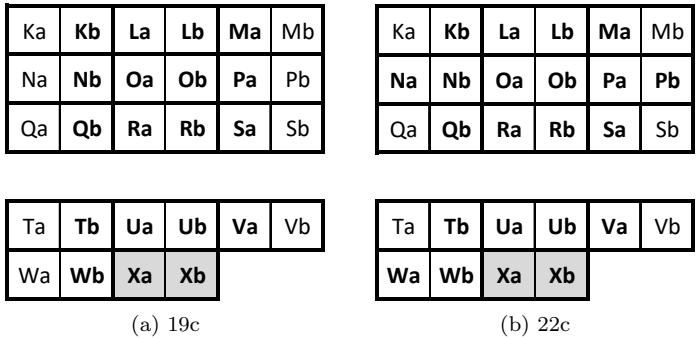
#### 3D Block-Based Masks

As discussed in previous works [57, 29], using a block-based approach allows to reduce the number of pixel look-ups in the first scan, and the amount of label merges required to complete the task. This leads to a reduction in the number of load/store operations required, and to a consequent performance improvement. Therefore, applying a block-based approach also to the 3D space is very likely to achieve a considerable improvement.

When considering 3D volumes, it is possible to define many different block-based masks with increasing complexity. If we think about blocks of voxels, the most simple and promising options are $2 \times 1 \times 1$, $2 \times 2 \times 1$ and $2 \times 2 \times 2$ blocks.

As already mentioned, one of the core principles on which block-based CCL algorithms are based on is that all pixels/voxels inside a block are connected. This means that blocks with more that 2 pixels/voxels per edge would have no benefit.

Considering that a 3D mask would contain 14 blocks in total (9 on the previous plane, 5 on the current plane), the full masks of the aforementioned block configurations would contain $14 \times (2 \times 1 \times 1) = 28$, $14 \times (2 \times 2 \times 1) = 56$, and $14 \times (2 \times 2 \times 2) = 112$ voxels respectively.




(a) 19c (b) 22c

Figure 4.15: 3D scan masks based on $2 \times 1 \times 1$ blocks. These masks consider respectively (a) 19 and (b) 22 conditions, which are highlighted in bold. Both masks spread over two slices: 19c mask contains 7 pixels on the current slice (bottom) and 12 pixels in the upper one (top), while 22c mask contains 8 pixels on the current slice (bottom) and 14 pixels in the upper one (top). Gray squares identify the current pixels that need to be labeled using the information extracted from their neighbors. Non-bold pixels are ignored, as they cannot cause any connection that is not already detectable through other pixels.

However, the idea is to slowly approach bigger masks, due to the explosion in complexity. Therefore, we start by considering masks containing only essential voxels, by simply removing those that are not direct neighbors of any $x$ voxel. Thus, for the $2 \times 1 \times 1$ block configuration a mask with 19 conditions, hereinafter called 19c, can be generated (Fig. 4.15a). The mask based on $2 \times 1 \times 1$ blocks could be extended up to 28 pixels. Anyway, the corner voxels Ka, Mb, Qa, Sb, Ta, Vb are never directly responsible for *merges* or *assignments*. They can thus be ignored to reduce the DTree generation complexity with no other side effects (Fig. 4.15b); the resulting mask is called 22c. A similar consideration can be applied to the other block configurations. As an example, the simplified version of the $2 \times 2 \times 1$ block-based mask (26c) is reported in Fig. 4.16.

Table 4.4: Entropy and information gain for the generation of the SAUF tree using the temporal popularity classifier. Each row corresponds to one node of the tree. Values have been rounded for visualization reasons. Bold values identify the entropy and the $IG$ of the condition chosen for the current node in the decision tree. The final resulting tree is reported in Fig. 4.17a.

| Node | Depth | $H(S)$ | p | | | q | | | r | | | s | | | x | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $H(T_0)$ | $H(T_1)$ | $IG$ | $H(T_0)$ | $H(T_1)$ | $IG$ | $H(T_0)$ | $H(T_1)$ | $IG$ | $H(T_0)$ | $H(T_1)$ | $IG$ | $H(T_0)$ | $H(T_1)$ | $IG$ |
| 1 | 0 | 2.2 | 2.0 | 1.4 | 0.5 | 2.3 | 1.5 | 0.3 | 1.9 | 2.1 | 0.2 | 2.1 | 2.1 | 0.1 | **0.0** | **2.4** | **1.0** |
| 2 | 1 | 2.4 | **2.0** | **0.8** | **1.0** | 2.5 | 1.0 | 0.7 | 1.8 | 2.3 | 0.4 | 2.2 | 2.2 | 0.2 | | | |
| 3 | 2 | 2.0 | | | | **2.0** | **0.0** | **1.0** | 1.5 | 1.5 | 0.5 | 1.5 | 1.5 | 0.5 | | | |
| 4 | 2 | 0.8 | | | | **1.0** | **0.0** | **0.3** | 0.0 | 1.0 | 0.3 | 0.8 | 0.8 | 0.0 | | | |
| 5 | 3 | 2.0 | | | | | | | **1.0** | **1.0** | **1.0** | 1.0 | 1.0 | 1.0 | | | |
| 6 | 3 | 1.0 | | | | | | | **0.0** | **0.0** | **1.0** | 1.0 | 1.0 | 0.0 | | | |
| 7 | 4 | 1.0 | | | | | | | | | | **0.0** | **0.0** | **1.0** | | | |
| 8 | 4 | 1.0 | | | | | | | | | | **0.0** | **0.0** | **1.0** | | | |

### Learning Decision Trees from OR-Decision Tables

With some adjustments, it is possible to apply the principles of decision tree learning to binary image processing. As discussed in Section 4.2, binary image processing algorithms can be described through the command execution metaphor by means of an OR-decision table.

Considering this table as a "dataset" and treating the actions as the *result classes*, it is possible to apply decision tree learning. This approach will allow the generation of a DTree that maps commands to actions, ideally checking the minimum number of conditions.

However, classifying the rule data poses a challenge: given three rules with actions $\{3\}$, $\{3, 4\}$ and $\{4\}$, which *classes* exist? Rule 1 and 2 are "equivalent", because they have a common intersection. However, rule 2 and 3 are also equivalent, but not rule 1 and 3, *i.e.* equivalence in this case is not transitive. If all rules remaining after a *split* are of the same class, a leaf is created. When only the two rules $\{\{3\}, \{3, 4\}\}$ or $\{\{3, 4\}, \{4\}\}$ remain after a *split*, a leaf can be created. However, the rules $\{\{3\}, \{3, 4\}, \{4\}\}$ can not be reduced to a single action. Hence, taking unique action combinations as classes, *e.g.* through a bitmapped representation, would not resolve cases like $\{\{3\}, \{3, 4\}\}$ correctly and would require a large amount of memory due to the high number of classes.

Therefore, a heuristic for the classification has been employed: for rules with more than one action, the action with the previous highest occurrence, *i.e.* the *temporal* "most popular" action, determines the "class" of this rule.

Another heuristic can be applied as well: calculating all single action occurrences in a separate pass, and always classifying based on the global "most popular" action. If there is a tie in popularity, the action with the lower id is taken in both classifiers. The global classifier has been found to be the most effective at obtaining high-quality decision trees, close to the optimal decision trees. The temporal popularity classifier performs slightly worse, but it is much faster since it does not require an additional pass for counting global rules. Using the temporal classifier, the first two rules of $\{\{3\},\{3,4\},\{4\}\}$ would be therefore classified in class 3, and the third rule in class 4, resulting in an entropy of 0.92 ($P_3 = \frac{2}{3}$, $P_4 = \frac{1}{3}$).

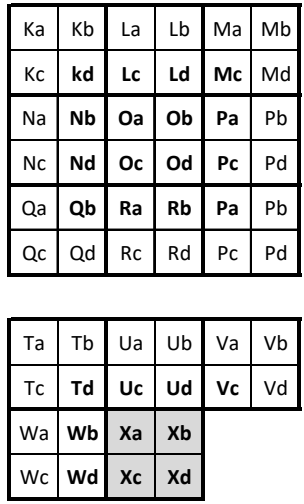



Figure 4.16: 3D scan masks based on $2 \times 2 \times 1$ blocks. This mask considers 26 conditions, which are highlighted in bold. The mask spreads over two slices: 10 pixels on the current slice (bottom) and 16 pixels in the upper one (top).

In order to demonstrate the application of decision tree learning to CCL and to binary image processing problems in general, a small example based on the Rosenfeld mask is discussed in the following of this Section. The example is based on $IG$ and *temporal* popularity classifier, and the step-by-step execution is reported in Table 4.4.

The Rosenfeld mask (Fig. 4.2b) has five conditions, and thus 32 different mask configurations (Fig. 4.3). For the first node, there are 5 conditions on which the split can be performed: $p$, $q$, $r$, $s$ and $x$. Splitting on these conditions results in the information gain values displayed in the first row of Table 4.4. Since $x$ has the highest information gain, a condition node with $x$ is created, and two child nodes are added. Because all rules with $x = 0$ result in action 1 (*no action*), the entropy is zero and the left child node is a leaf. On the other hand, rules with $x = 1$ do not have a common intersection and the entropy is greater than zero. This results in a non-leaf node. The procedure is repeated in the child nodes until all leaves are created (Fig. 4.17a). As can be noticed from the comparison with the optimal DTree (Fig. 4.17b) obtained using

the algorithm presented in [72], in this specific case the heuristic-generated tree contains only one extra node.

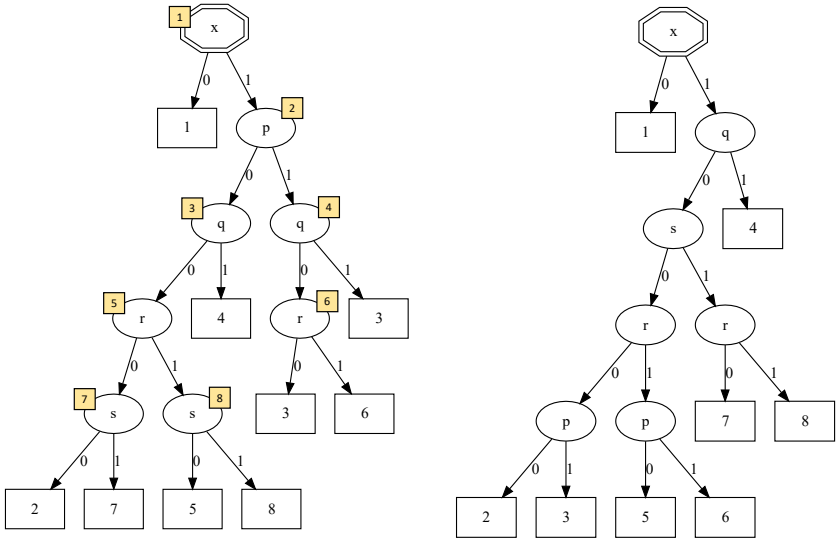
### 4.6.3 Experimental Results

Algorithms generated with the proposed strategy are evaluated using YACCLAB [80, 78, 79], the already mentioned open source *C++* benchmarking framework for CCL algorithms. Experimental results discussed in the following are obtained on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. All the algorithms have been compiled for x64 architecture with optimizations enabled.

Exploiting the DTree learning strategy presented in Section 4.6.1, we generated three different algorithms: EPDT_19c, EPDT_22c, and EPDT_26c. They exploit a two scan approach based on the $2 \times 1 \times 1$ (19c, 22c) and the $2 \times 2 \times 1$ (26c) block-based masks. Obtained DTree have also been compressed as described in [60] to minimize code footprint. The proposed heuristic solutions have been compared to state-of-the-art algorithms for 3D CCL: Label-Equivalence-Based (LEB) and Run-Based Two-Scan (RBTS), by He *et al.* [116].

Table 4.5 reports average execution times of the considered algorithms over the three datasets. In order to discover which label equivalence solving method works best for the EPDT algorithms, four different options have been considered: standard Union-Find (UF), Union-Find with Path Compression (UFPC) [31], Three Table Array (TTA) [67], and interleaved Rem algorithm with SPlicing (RemSP) [41]. For what concerns LEB and RBTS, instead, the TTA solver is always employed since it delivers the best results according to [116].

In order to achieve a deeper understanding of execution time distribution, two more tests have been performed. Only one label solver has been considered for EPDT algorithms: RemSP, which, according to Table 4.5, on average carries out the best performance. Bar charts of Fig. 4.18 report average execution time split between the three steps composing the algorithms: memory allocation and deallocation, first scan, and second scan. Finally, Table 4.6 compares the amount of average memory accesses on the binary image, labels image and equivalence vector of pixel- and block-based algorithms. RBTS, having a considerably different memory access pattern due to its run-based nature, has been excluded.




(a) EPDT (b) Optimal DTree

Figure 4.17: (a) decision tree generated using *IG* and temporal popularity classifier for the rules given in Fig. 4.3. The entropy values used for the creation of this tree are reported in Table 4.4, with yellow numbers indicating the respective node numbers. (b) one of the optimal decision trees that can be generated with the algorithm described in [29] starting from the decision table of Fig. 4.3. Ellipsis represent the conditions to be checked, and rectangles (leaves) contain the actions to be performed, which are identified by integer numbers. (1) $x \leftarrow no\,action$, (2) $x \leftarrow new\,label$, (3) $x \leftarrow p$, (4) $x \leftarrow q$, (5) $x \leftarrow r$, (6) $x \leftarrow s$, (7) $x \leftarrow p + r$, (8) $x \leftarrow r + s$.

The comparison of raw execution times in Table 4.5 demonstrates that the best EPDT algorithm is 22c. It also outperforms LEB, the current state-of-the-art, on Hilbert and Mitochondria, with an average speed-up of 1.25× and 1.15× respectively, while obtaining the same performance on OASIS.

As can be noticed, EPDT_19c is better than EPDT_26c, but worse than EPDT_22c in all test cases. In order to explain this performance

gap, several factors must be taken into account. Among them, the most significant are the number of load/store operation, the amount of *merge* required, and code size.

Table 4.5: Average Results in ms. Lower is better.

| | **Hilbert** | **Mitochondria** | **OASIS** |
|---|---|---|---|
| EPDT_19c_RemSP | 4.77 | 276.72 | 30.12 |
| EPDT_19c_TTA | 5.07 | 281.07 | 30.54 |
| EPDT_19c_UF | 4.78 | 277.66 | 30.38 |
| EPDT_19c_UFPC | 4.81 | 277.63 | 30.57 |
| EPDT_22c_RemSP | 4.68 | 276.48 | 29.07 |
| EPDT_22c_TTA | 4.84 | 276.66 | 29.00 |
| EPDT_22c_UF | 4.73 | 276.79 | 29.16 |
| EPDT_22c_UFPC | 4.69 | 271.62 | 29.36 |
| EPDT_26c_RemSP | 7.96 | 398.39 | 51.22 |
| EPDT_26c_TTA | 8.15 | 398.30 | 51.45 |
| EPDT_26c_UF | 8.02 | 400.45 | 51.61 |
| EPDT_26c_UFPC | 7.93 | 399.57 | 51.62 |
| LEB_TTA | 5.84 | 313.63 | 29.03 |
| RBTS_TTA | 8.24 | 430.46 | 45.50 |

EPDT_22c considers all the meaningful pixels of the $2\times1\times1$ block-based mask, adding also the pixels ignored by EPDT_19c. Including missing pixels and generating the corresponding DTree allows to reduce the total number of load/store operations required by the algorithm from $\sim$1% to $\sim$3%, depending on the considered dataset. Indeed, using the information provided by additional pixels, it is possible to identify equivalence classes earlier during the first scan, without having to perform expensive merge operations later.

Nevertheless, expanding the size of the mask has a cost: the generated tree code lines increase from $\sim 4k$ for EPDT_19c to $\sim 7k$ for EPDT_22c, negatively impacting the instruction cache. However, the benefits of reducing merges and in general load/store operations outclass the disadvantages of the increased cache miss rate.

On the other hand, when comparing EPDT_19c and EPDT_26c we face a totally different scenario: the reduction in load/store operations is

Table 4.6: Average number of load/store operations on the *OASIS* dataset. Quantities are given in millions.

| **Algorithm** | **Binary Image** | **Labels Image** | **Equivalences Vector** | **Total** |
|---|---|---|---|---|
| LEB | 11.461 | 27.182 | 9.851 | 48.494 |
| EPDT_19c | 14.917 | 17.760 | 1.169 | 33.846 |
| EPDT_22c | 14.057 | 17.753 | 1.145 | 32.955 |
| EPDT_26c | 13.695 | 13.145 | 0.728 | 27.568 |

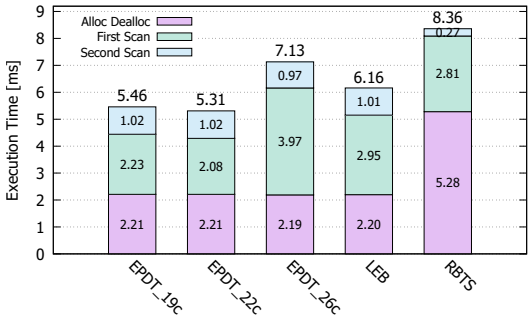
much more significant ($\sim$10% $\div$ $\sim$19% depending on the dataset), but at the same time the tree size increases up to more than $41k$ lines of code. In this case the cache miss rate prevails, causing an overall performance degradation. The described effects are magnified by any further increase in the mask size. For this reason, we limit our analysis to masks up to 26 conditions.

Comparing EPDT_22c with LEB, conclusions similar to those above can be drawn. LEB is based on a hand-crafted decision tree that guarantees the minimum number of accesses to the input image, while neglecting what concerns merges and accesses to the output image. Taking the OASIS dataset as a reference (Table 4.6), LEB requires about $1.4x$ the load/store operations of EPDT_22c, on average. On the other hand, the small tree employed by LEB completely fits instruction cache: EPDT_22c and LEB have comparable performance on OASIS dataset. As the number and the complexity of connected components grow (Mitochondria and Hilbert), the gap in the number of load/store operations and the improvement of EPDT_22c over LEB become more evident.

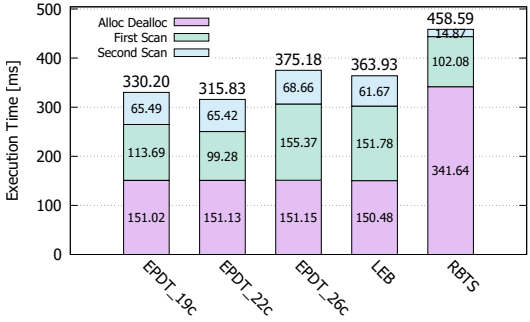
Differentiating time spent in actual algorithm execution from time used for memory allocation (Fig. 4.18), it is possible to draw further interesting conclusions. EPDT_19c, EPDT_22c, EPDT_26c, and LEB make use of the same data structures, taking the same time to allocate and deallocate memory (negligible oscillations are due to measurement accuracy). In contrast, RBTS requires additional data structures to keep track of run information, thus spending about twice the time in memory allocation. On the other hand, the additional information stored by RBTS allows to significantly reduce the number of merge operations in the first scan, and to early identify final labels, avoiding to replace provisional ones during

the second scan. Obviously, datasets with few connected components and few patterns that cause merge occurrence (OASIS) practically nullify the advantages of runs.

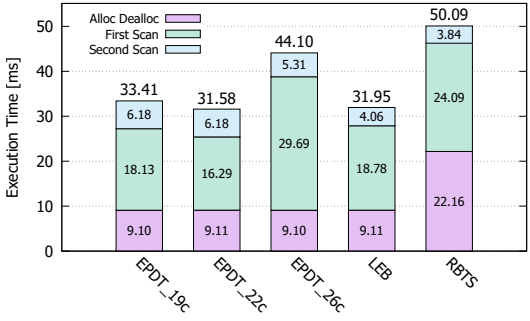
However, the benefits introduced by runs are not sufficient to compensate for the allocation cost overhead. For this reason, RBTS is the worst in all considered settings. In very specific context, where it is realistic to perform memory allocation only once, like an embedded system which capture fixed size images, RBTS would be one of the best performing solution.




(a) Hilbert




(b) Mitochondria




(c) OASIS

Figure 4.18: Average run-time tests with steps in ms (lower is better).

## 4.7 Conclusion

In this Chapter, we presented GRAPHGEN, a framework which allows to generate optimal decision trees, and to automatically apply state prediction, compression and path-length optimization based on frequencies for any binary image processing problem. The only requirement for the user is to model his needs as a set of rules. The effectiveness of all these features has been showcased on three different common binary image processes. Furthermore, implementations for usage in parallel GPU environments and for 3D algorithms have been demonstrated. Finally, a novel approach has been proposed for generating heuristic —instead of optimal— decision trees starting from a decision table. The resulting approach, Entropy Partitioning Decision Tree, EPDT in short, has been proven to generate near-optimal decision trees when applied to connected components labeling. Through the proposed heuristic it was possible to extend the block-based approach to 3D volumes, significantly improving the state-of-the-art on the field.

# Chapter 5

# Connected Components Labeling on Bitonal Images

Binary images can be efficiently stored with only 1 bit per pixel ("1-bit graphics" format, or bitonal images). This representation is especially useful in embedded systems with limited resources, where memory usage must be reduced to a minimum. In banking, as an example, the bitonal image is the legally recognized standard for electronic check clearing in the United States and many other countries. Working with 1-bit per pixel images (also denoted as 1-bpp or bitonal images) allows to considerably reduce the amount of memory accesses; on the other hand, it also requires additional bitwise operations for retrieving single pixel values.

This chapter discusses connected components labeling on bitonal images. Two proposals are presented, which come from completely different approaches: the first is derived from the application of GRAPHGEN, described in the previous Chapter, to blocks of 4x1 bits, while the second exploits compiler intrinsics to efficiently find consecutive blocks of foreground pixels, also known as runs.

## 5.1 Connected Components Labeling on Bitonal Images

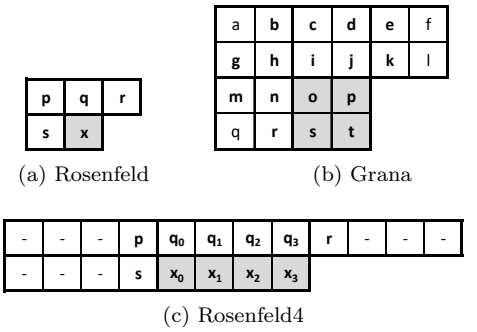
This Section proposes a systematic approach for labeling multiple pixels at once, automatically generating the actions to be performed on the current pixel/block given the mask values. The proposed strategy allows to extend existing techniques for the generation of optimal decision trees to much more complex masks, where the connectivity between pixels inside a block is not guaranteed. A showcase application, consisting in the design of an efficient CCL algorithm for bitonal images, demonstrates the effectiveness of our proposal in terms of speed and memory footprint. In the context of 1-bpp images, being able of labeling an entire byte as a single block would guarantee a significant performance improvement, without requiring to convert the input into a 1-byte per pixel image. Unfortunately, the assumption that foreground pixels are always connected inside a block does not hold in such a case, and algorithms proposed in literature for the automatic generation of binary decision trees are not feasible. This Section introduces a reliable method for generating all the possible actions associated to a scanning mask, which is employed to design an extremely fast CCL algorithm for bitonal images capable of labeling four consecutive pixels at once.

### 5.1.1 Method

In this Section, the proposed method for labeling multiple pixels at once is presented. As usual, CCL is performed with two raster scans of the input, here briefly summarized.

The first scan employs a mask moving in discrete steps, which highlights the current pixel(s) to be labeled and its neighborhood, composed of already analyzed and labeled pixels; at each step, the current pixel is assigned a provisional label, and if it connects two or more connected components, their labels are recorded as equivalent by means of some variation of union-find. This set of operations carried out for a certain mask position is known as *action*, and depends on the values of pixels inside the mask, which form a binary word known as *command* [29].

Then, the second scan simply replaces each provisional label with the chosen representative for the equivalence class, thus completing the task. While the second scan is usually fixed and nearly identical for most al-




(a) Rosenfeld (b) Grana

(c) Rosenfeld4

Figure 5.1: Example of scan masks. Gray squares identify current pixels to be labeled using information extracted from white pixels. (a) and (b) are very common masks employed in CCL (c) is an extended version of the Rosenfeld mask that is proposed and analyzed in this paper. In this specific case, the current pixels, i.e. $x_0$, $x_1$, $x_2$, $x_3$, do not necessarily share the same label. Dashes identify meaningless pixels.

gorithms, the first scan is where algorithmic proposals differ the most: here several optimizations can take place, such as the aforementioned decision tree (decide the action without reading the whole command word), block-based strategy (label multiple connected pixels at once), prediction (avoid to re-read neighbor pixels known from the previous step), and compression (reduce machine code size by merging equivalent subtrees of a larger decision tree).

The new technique proposed with this work extends the block-based approach, by overcoming the limitation that all pixels to be labeled at once must be connected. In fact, with respect to previous proposals [69, 120, 29], limited to a block size of at most $2 \times 2$ pixels, the devised algorithm can be applied to blocks of any shape.

In the following, the term *macroblock* identifies the pixels of the mask to be labeled during the current step. A macroblock can be divided into disjoint *blocks*, each of which always contains pixels connected to each other. This ensures that a block can always be assigned only one label.

On the other hand, it is not possible to assume that only one single action is performed on the current macroblock. Taking the mask of Fig. 5.1c

as an example, the macroblock is composed by pixels $x_0$, $x_1$, $x_2$, and $x_3$ which can be divided into two different blocks: $x_0, x_1 \in X_0$ and $x_2, x_3 \in X_1$.

In this context, it may happen that, e.g., block $X_0$ requires a *new label*, while $X_1$ must be assigned the result of a *merge* —the union procedure of the union-find data structure— of two different existing label classes.

In literature, no attempt has been made to systematically generate the set of actions associated to a macroblock-based scan mask. The block-based mask proposed in [29] (Fig. 5.1b), for example, shares the same actions of the Rosenfeld mask, with the only addition of some merge operations that have no effect in a pixel-based context.

Let us start with some formal definitions. Be $I : \mathcal{L} \to \{B, F\}$ a binary image, *i.e.*, a function defined over a 2-dimensional square lattice $\mathcal{L}$, where pixels only assume two possible values, background ($B$) and foreground ($F$), usually represented by integers 0 and 1 respectively.

The 8-neighborhood of pixel $p = (p_r, p_c) \in \mathcal{L}$, denoted as $\mathcal{N}_8(p)$, is the set of pixels sharing an edge or vertex with $p$:

$$\mathcal{N}_8(p) = \{q = (q_r, q_c) \in \mathcal{L} \mid max(|p_r - q_r|, |p_c - q_c|) \leq 1\} \tag{5.1}$$

Given $S \subset \mathcal{L}$, pixels $p, q \in S$ are *connected in* $S$, denoted as $p \diamond_S q$, if a path of neighbor foreground pixels exists, all belonging to $S$ and leading from $p$ to $q$:

$$p \diamond_S q \Leftrightarrow \exists \{s_i \in S \mid I(s_i) = F \wedge s_1 = p, s_{n+1} = q, s_{i+1} \in \mathcal{N}_8(s_i), \forall i = 1, ..., n\} \tag{5.2}$$

Connectivity in $S$ is an equivalence relation, since the properties of *reflexivity*, *symmetry* and *transitivity* hold. Equivalence classes of this relation are called *Connected Components (CCs)* of $S$. When $S = \mathcal{L}$, we omit the subscript in the notation $p \diamond q$, and CCs of $\mathcal{L}$ are referred to as just CCs.

To better detail the proposed algorithmic solution, we divide the pixels in the mask in two subsets:

- *Outer Pixels ($P_O$)*, pixels inside the mask but outside the macroblock. Outer pixels already have a provisional label, since they have already been analyzed by the algorithm.
- *Inner Pixel ($P_I$)*, pixels inside the macroblock. Inner pixels must be assigned a provisional label in the current step.

As an example, in Fig. 5.1c, $P_O = \{p, q_0, q_1, q_2, q_3, r, s\}$, while $P_I = \{x_0, x_1, x_2, x_3\}$. In order to proceed with the generation of the action set, the following operations are required for each configuration of the mask:

- Identify the CCs of $P_O$; this set of outer connected components is denoted as $C_O$;
- Identify connected components of $P_I$; these inner connected components are denoted as $C_I$;
- Identify the connected components for the whole mask, i.e., CCs for $P_O \cup P_I$; these are denoted as $C_T$;
- For each $t \in C_T$, consider all inner pixels belonging to $t$ (i.e. the set $t \cap P_I$); all these pixels must be assigned the same label $l$. This label is determined analyzing the set of outer connected components contained in $t$, denoted as $C_O^t = \{c \in C_O \mid c \subset t\}$. If $C_O^t = \varnothing$, then a new label is created for $l$. If $\#C_O^t = 1$, then $l$ can be assigned the label of any pixel of $c \in C_O^t$. Finally, if $\#C_O^t > 1$, all CCs in $C_O^t$ must be merged, meaning that their labels are marked as equivalent and $l$ is set to any of them (typically the smallest).

It is important to stress that $C_T \neq C_O \cup C_I$, and that the external components are defined without considering connections through pixels currently under examination (the pixels of the macroblock). The same goes for internal components, where we do not consider connections due to external pixels. Those connections are considered only for $C_T$.

An example of action generation is reported in Fig. 5.2. In Fig. 5.2a, a mask configuration is shown: the gray squares represent foreground pixels, $x_0$, $x_1$, $x_2$, $x_3$ are the pixels in the "current" macroblock, and dashes identify meaningless pixels. The process starts with the detection of the connected components in the outer part of the mask, i.e. ignoring the current macroblock (Fig. 5.2b). In this specific example, three different objects are in $C_O$, respectively named 1, 2, 3. Then, inner connected components are identified inside the macroblock: $a$ and $b$ as in Fig. 5.2c. Finally global connected components ($C_T$), $x$ and $y$, are depicted in Fig. 5.2d. In particular, CC $x$ contains both the inner component $a$ and the outer component 1, so from this configuration we can derive the first operation to be performed, $a = 1$, that easily translates into action $x_0 = p$. This action

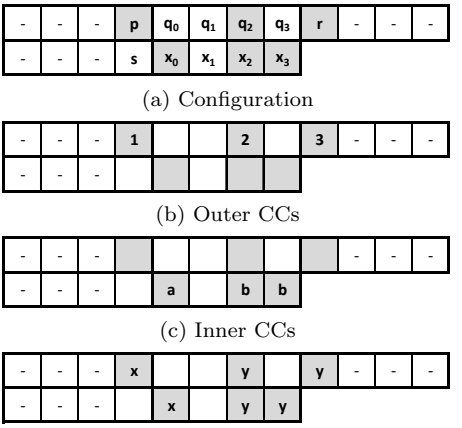



Figure 5.2: Example of the proposed action(s)-generation algorithms when applied to the mask configuration depicted in (a). Gray squares identify foreground pixels inside the mask. The pixels to be labeled using the information extracted from all the others are $x_0$, $x_1$, $x_2$, and $x_3$. Together they constitute the inner part of the mask. Remaining pixels are the outer part of the mask. Dashes identify meaningless values.

means: *assign to* $x_0$ *the same label previously assigned to* $p$. Moreover, CC $y$ contains inner component $b$ and outer components 2 and 3. In this case the operation is $b = 2 + 3$: *assign to all the pixels of connected component* $b$ *the result of the merge between components* 2 *and* 3. Translating this operation into an action, we obtain $x_2\ x_3 = q_2 + r$, that is *assign to pixels* $x_2$ *and* $x_3$ *the merge of labels previously assigned to pixels* $q_2$ *and* $r$.

In order to give the reader an additional example, we can consider the same configuration of Fig. 5.2a and add $q_1$ as foreground. In this case, outer and inner components are the same of Fig. 5.2b and Fig. 5.2c, but component 2 is made of two pixels ($q_1$ and $q_2$) instead of just one. On the other hand, we obtain just one single global component. This causes the algorithm to generate the action $x_0\ x_2\ x_3 = p + q_1\ q_2 + r$: $x_0$, $x_2$, *and* $x_3$ *must be assigned the result of the merge between* $p$, *one between* $q_1$ *and* $q_2$, *and* $r$. Actually, the *or* between $q_1$ and $q_2$ is responsible for the generation of two equivalent actions, $x_0\ x_2\ x_3 = p + q_1 + r$ and

$x_0\ x_2\ x_3 = p{+}q_2{+}r$. Most equivalence cases can be resolved with the block-based approach described in the following; the others are treated during the generation of the optimal decision tree, as explained in [72]. Basically, each global connected component generates one or multiple equivalent actions, responsible for the labeling of all pixels belonging to one or more internal components.

### Reducing Actions with Blocks

The number of actions generated with the proposed approach grows very quickly as the mask size increases, making the generation of the optimal decision tree extremely hard or even impossible. In order to reduce the number of actions and simplify the problem, we introduce a block-based approach. As described above, macroblocks are divided into blocks, and pixel-based actions are replaced with block-based ones, eliminating possible duplicates. This way, many of the previous actions translate into the same one, and can be removed.

As an example, let us consider the following pixel-based actions: $x_0 = q_0$ and $x_1 = q_0\ q_1$, the latter actually representing the two equivalent actions $x_1 = q_0$ and $x_1 = q_1$. Since $x_0$ and $x_1$ are always connected, they can be viewed as part of the single block $X_0$, and the same applies to $q_0$ and $q_1$, which are part of the block $Q_0$. Thus all the three aforementioned pixel-based actions can be fused into $X_0 = Q_0$, producing the same outcome.

As previously described in literature [29, 102], when working with block-based actions the second scan of an algorithm requires a slight overhead to correctly handle blocks, i.e. assigning the block label to all foreground pixels belonging to that block. Obviously, the reduction in the number of actions can be more or less significant depending on the mask features. For what concerns the mask of Fig. 5.1c, actions reduce of about 80% (from 413 to 85 actions) when moving to the block-based approach. After generating all the possible actions associated to a scan mask and the corresponding *OR*-decision table, the algorithm described in [72] can be employed for the generation of an optimal decision tree, which maps the mask configuration to an action, minimizing the average number of load/store operations required.

The described approach has been employed to generate an optimal decision tree for the mask of Fig. 5.1c, which constitutes the core of a new CCL algorithm, specifically designed for 1-bpp images. Since it shares

the general structure of SAUF, but operates on 4-pixel macroblocks, it is referred to as SAUF4_1BPP.

### 5.1.2 Experimental Results

The performance evaluation of the proposed algorithm has been carried out with an extended version of the YACCLAB benchmark [80, 79]. In order to incorporate a standard well-known implementation of bitonal images, the benchmark has been integrated with Leptonica, an open-source image processing library employed in several projects (e.g. Tesseract OCR by Google). The extended version of the YACCLAB benchmark, including the proposed algorithm implementation, is available at [78].

Experimental results discussed in the following were obtained on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. Our proposal is evaluated on three datasets: *Fingerprints*, *Medical* and *Tobacco800*, which cover the most common CCL application fields, a full description can be found in [62]. Fig. 5.3 highlights how the performance of algorithms is influenced by the different phases they are composed of: memory management, first scan and second scan.

The selected algorithms for comparison are SAUF, BBDT, Spaghetti, and the CCL implementation available in Leptonica. The first three algorithms, mentioned in Section 3.1, represent the state of the art regarding 1-byte per pixel images; for a fair comparison, their first scan times also include a conversion of the input to their preferred format. SAUF_1BPP and SAUF4_1BPP, finally, are the 1-bpp algorithms introduced in this paper. The former is a simple adaptation of SAUF, which iterates over the eight pixels stored in each byte; the latter employs a decision tree generated starting from the mask of Fig. 5.1c, employing the action generation algorithm introduced with this paper. All the algorithms employ the classic union-find label solver [31].

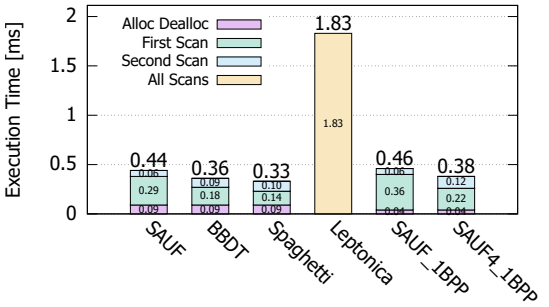
The Leptonica algorithm is based on a seedfill approach which, as can be observed, is extremely inefficient when connected components extend vertically (e.g. *Fingerprints*), causing a series of non cache-friendly memory accesses. On the other hand, when small size CCs constitute the images, Leptonica has comparable performance with SAUF_1BPP.

The main proposal of this work, SAUF4_1BPP, considerably exceeds the performance of Leptonica, with a speedup ranging from 1.13 to 4.81

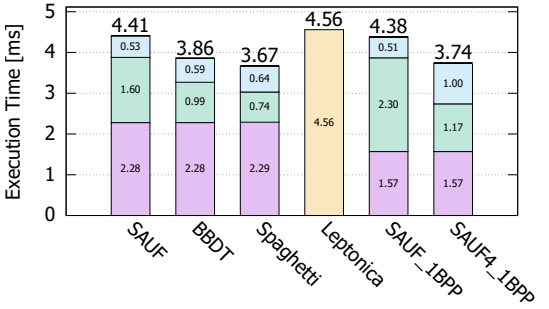
depending on the dataset, and thus represents the currently most efficient CCL algorithms designed to work on bitonal images. Moreover, SAUF4_1BPP has comparable performance to Spaghetti (current state of the art for CCL on binary images), when the latter needs a prior conversion of the input. However, SAUF_1BPP only requires about $1/9\times$ memory for the input data, making it an excellent choice for use cases where memory size is constrained.

### 5.1.3 Conclusion

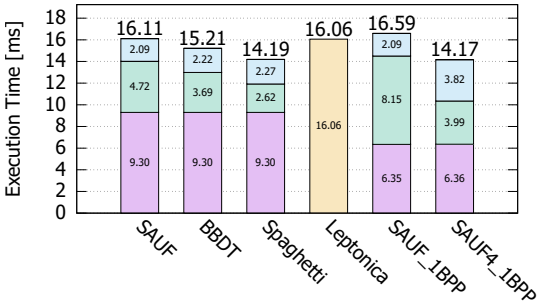
An effective solution to automatically map a connected components labeling scan mask configuration with the actions to be performed has been presented, which allows to label multiple pixels at each mask shift. A CCL algorithm generated using this systematic approach is presented, which outperforms competitors on bitonal images, confirming the effectiveness of the method.




(a) *Fingerprints*




(b) *Medical*




(c) *Tobacco800*

Figure 5.3: Average run-time tests with steps in ms (lower is better).

## 5.2 Fast Run-Based Connected Components Labeling for Bitonal Images

This Section extends an existing run-based CCL strategy, adapting it to the bitonal format, and optimizes it with *Find First Set* hardware operations and a smart management of provisional labels; the result is an efficient solution called Bit-Run Two Scan (BRTS). Then, BRTS is further optimized by merging pairs of consecutive lines through bitwise *OR*, and finding runs on this reduced data. This modification is the basis for another new algorithm on bitonal images, Bit-Merge-Run Scan (BMRS). When evaluated on a public benchmark, the two proposals outperform all the fastest competitors in literature, and therefore represent the new state of the art for connected components labeling on bitonal images.

Our work starts from the observation that the combination of the bitonal image format with *Find First Set* instructions allows to efficiently retrieve consecutive blocks of connected pixels, also known as *runs*. *Find First Set* (FFS), also called *Count Leading Zeros*, *Bit Scan Forward* or *Find Leftmost One*, is a bitwise operation that reports the position of the first bit set to 1 in a word, counting from the least significant bit; this is an especially efficient operation because it is implemented in hardware on most recent architectures.

The run-based approach has been applied to CCL before, by He *et al.* [40]. Their proposal, denoted as Run-Based Two-Scan (RBTS), employs the typical two-scan approach, but considers runs as the elementary units of connected components, instead of single pixels or blocks. In this work, RBTS is modified to work with bitonal input, and improved with two more optimizations: FFS instructions, and a more efficient method to store provisional labels. The resulting algorithm is called Bit-Run Two Scan (BRTS).

Then, we noticed that runs computed on the bitwise *OR* between consecutive rows directly correspond to connected components in the original data (Fig. 5.4). This observation led to the design of another new algorithm, Bit-Merge-Run Scan (BMRS), which finds runs on the bitwise *OR* between pairs of rows, also denoted as *merged runs*. This approach requires a specific method to check connectivity between merged runs, but approximately halves the number of runs to be computed.

The two proposals are evaluated on a public benchmark, and com-

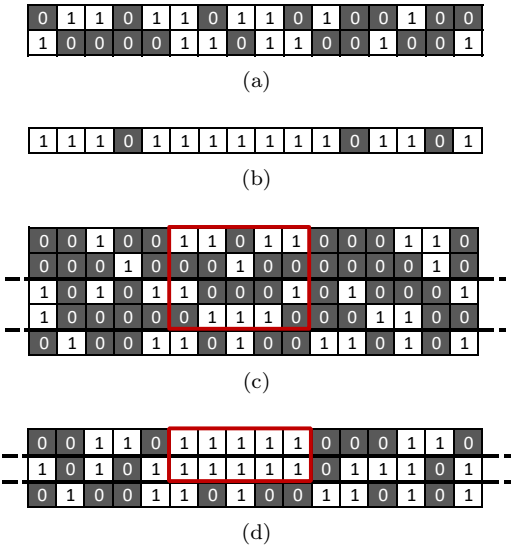



Figure 5.4: (a) Example rows to be merged into a single one by means of bitwise *OR*. The image contains four connected components, which corresponds to runs after the merge, whose result is reported in (b). (c) Example image with five rows, that are merged into three rows in (d). Merged runs enclosed in the red rectangle appear to be connected in (d), but original data in (c) show that they are not. Best viewed in color.

pared to the fastest CCL algorithms available in literature. Experimental results demonstrate that, when fed with bitonal input, BRTS and BMRS outperform all the competitors, establishing the new state-of-the-art for connected components labeling on bitonal images.

### 5.2.1 Background: Run-Based Two-Scan (RBTS)

The first of the two CCL algorithms proposed with this paper, BRTS, is a special optimization of Run-Based Two-Scan (RBTS), the run-based algorithm devised by He *et al.* in [40], which is described in this Section.

As the name suggests, the algorithm is built upon the concept of *run*, which is a block of contiguous foreground pixels in a row of $\mathcal{L}$. It is clear

that a run is always a subset of a connected component, and that each connected component can be split in a finite set of disjoint runs.

A run starting at pixel $p = (r, c_s)$ and ending with pixel $q = (r, c_e)$ is denoted as $\rho = \mathscr{r}(r, c_s, c_e)$. The *neighborhood of a run* is the union of the neighborhoods of all the pixels in the run:

$$\mathcal{N}_8(\rho) = \bigcup_{p \in \rho} \mathcal{N}_8(p)$$

For $\rho = \mathscr{r}(r, c_s, c_e)$, this corresponds to:

$$\mathcal{N}_8(\rho) = [\![r-1, r+1]\!] \times [\![c_s - 1, c_e + 1]\!] \cap \mathcal{L}$$

Moreover, we define the *upper neighborhood* of a run as the part of the neighborhood in the upper row:

$$\mathcal{N}_8^{\wedge}(\rho) = \{r-1\} \times [\![c_s - 1, c_e + 1]\!] \cap \mathcal{L}$$

The RBTS algorithm performs two scans of the input image $I$: the first one scans pixels in the raster scan direction, recording data for each run met, assigning provisional labels to runs and recording equivalences; then, the second scan replaces provisional labels with definitive ones.

During the first scan, when a run $\rho = \mathscr{r}(r, c_s, c_e)$ is found, its upper neighborhood is checked for the presence of connected runs, already recorded by the algorithm; $\mathcal{S}(\rho)$ is the set of these runs. Three possibilities can arise for $\mathcal{S}(\rho)$: *(i)* $\mathcal{S}(\rho) = \varnothing$, $\rho$ is assigned a new provisional label; *(ii)* $\mathcal{S}(\rho)$ only contains one run $\sigma$, the label of $\sigma$ is also given to $\rho$; otherwise, *(iii)* in the case that $\mathcal{S}(\rho)$ contains multiple runs, all of their labels are merged together, and the representative one of the resulting equivalence class is finally assigned to $\rho$. After the choice of the appropriate label $l$, the output image $L$ is updated so that $L(p) = l, \forall p \in \rho$.

The second scan replaces the label assigned to each pixel with the representative for the equivalence class, thus completing the labeling process. Optionally, this second scan can be preceded by a *flatten* operation on the label solver, which ensures that definitive labels are consecutive [31].

The first scan requires a method for finding the runs in the upper neighborhood of each new run $\rho$. In order to accomplish this, run metadata, consisting in start and end coordinates, are recorded in a *run queue*. A run $\sigma = \mathscr{r}(r-1, h, t)$ is connected to $\rho = \mathscr{r}(r, s, e)$ if $h \leq e + 1$ and

$t \geq s - 1$; therefore, when the raster scan reaches run $\rho = \mathscr{r}(r, s, e)$, any run $\sigma = \mathscr{r}(r - 1, h, t)$ with $t < s - 1$ can no longer be part of the upper neighborhood of any coming run, and its metadata cease to be useful. As a consequence, the run queue can be implemented as a circular buffer of size $(W/2 + 2)$.

### 5.2.2 Method

This Section introduces two new CCL algorithms, specifically designed for dealing with input in bitonal format, i.e., image $I$ is stored with only one bit per pixel, occupying approximately $(H \times W)/8$ bytes. This is the most memory-efficient way to store a binary image, and therefore represents a natural choice of format. The two proposed algorithms are called Bit-Run Two Scan (BRTS) and Bit-Merge-Run Scan (BMRS). In the following, bits in a 64-bit word are supposed to be stored with the leftmost pixel in the least significant bit and the rightmost one in the most significant one. If the image uses an opposite convention, one just needs to exchange Find First Set/Bit Scan Forward instructions with Find Last Set/Bit Scan Reverse ones.

#### Bit-Run Two Scan (BRTS)

This algorithm, as the acronym may suggest, is a special optimization of RBTS, which has been described in the previous section. The basic two-scan structure is inherited, together with the run-based nature and the use of a label solving technique for dealing with equivalence between provisional labels. The main improvement concerns the method to retrieve runs while scanning the input image, that in this case is stored in bitonal format.

Start and end position of runs are retrieved by means of *Find First Set (FFS)* instructions. Find First Set is a bitwise operation that, given an unsigned machine word, designates the position of the least significant bit set to one, counting from the least significant bit. Most modern CPU instruction set architectures provide FFS as a hardware operation, making it an efficient way to find start and end positions of runs. The run retrieval procedure is detailed in Algorithm 5.

Another optimization wrt RBTS is that provisional labels are not written in $L$, until the second scan. Instead, provisional labels are stored in

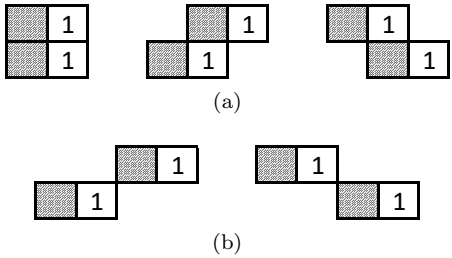



Figure 5.5: Examples of connectivity between two foreground pixels belonging to consecutive rows. Foreground pixels in the upper ($u$) and lower ($d$) row are denoted by 1, and squares filled with diagonal lines pattern are the added bits in expression $u \vee u \ll 1$ and $d \vee d \ll 1$. The result of $(u \vee u \ll 1) \wedge (d \vee d \ll 1)$ is non-zero for the three cases in (a), which are indeed 8-connected, and is zero for the two non-connected cases in (b).

an additional field of run metadata. In this way, each label is written in memory only once per run, instead of once per pixel, during the first scan. The downside of this approach is increased memory occupancy; in fact, run metadata must live until the end of the second scan, and so they cannot be stored in a circular buffer, as in RBTS. Because the worst case is that of alternate foreground and background pixels in each row, the maximum number of runs is $H \times (W/2 + 1)$. Metadata for each run include the the start and end columns, and the provisional label. For most real world applications, coordinates can be stored with 16 bits each and labels require 32 bits, so the total memory requirement for metadata amounts to $H \times (W/2 + 1) \times 8 \approx 4HW$ bytes, the same as the output image $L$. Total memory allocation for the algorithm is summarized in Table 5.1.

The second scan of BRTS writes definitive labels into $L$, using the metadata previously saved, and checking the correspondence between provisional and definitive labels with the chosen label solving strategy. Differently from RBTS, this correspondence is only checked once per run, instead of once per pixel; for this reason, this second scan is faster than that of RBTS.

### Bit-Merge-Run Scan (BMRS)

The second proposal of this paper, Bit-Merge-Run Scan (BMRS), is a modified version of BRTS.

**Algorithm 5** Runs retrieval algorithm used in BRTS and BMRS, described as a coroutine. Parameters *bits* and *bit_final* are pointers to the start and one past the end of the current row, and FFS is the FindFirstSet operation. Word size is assumed to be 64 bits and incrementing a pointer moves it to the next 64 bits.

```
 1: coroutine FindNextRun(bits, bit_final)
 2:     work_bits ← *bits
 3:     base ← 0
 4:     bitpos ← 0

 5:     loop
 6:         ▷ Find first non empty word
 7:         while FFS(&bitpos, work_bits) = 0 do
 8:             bits ← bits + 1
 9:             base ← base + 64
10:             if bits ≥ bit_final then
11:                 return (0xFFFF, 0xFFFF)
12:             work_bits ← *bits
13:         start ← base + bitpos
14:         work_bits ← ¬work_bits ∧ (¬0LL ≪ bitpos)

15:         ▷ Find ending position
16:         while FFS(&bitpos, work_bits) = 0 do
17:             bits ← bits + 1
18:             base ← base + 64
19:             if bits = bit_final then
20:                 bitpos ← 0
21:                 work_bits ← ¬0LL
22:                 break
23:             work_bits ← ¬(*bits)
24:         end ← base + bitpos
25:         work_bits ← ¬work_bits ∧ (¬0LL ≪ bitpos)
26:         yield (start, end)
```

In a two-row binary image $I : [\![0,1]\!] \times [\![0, W-1]\!] \to \{0,1\}$, a bitwise *OR* between the upper row and lower row yields connected chunks of foreground pixels (runs) that correspond to connected components in $I$: see the example in Fig. 5.4a and Fig. 5.4b.

The whole idea of BMRS is based on this property. A bitwise *OR* is performed for every disjoint pair of consecutive rows, and the result is saved in the *merged input*, which requires approximately $HW/16$ bytes. Then, runs are found on the merged bits; these runs will be referred to as *merged runs*, in opposition to runs computed directly on the raw input.

Unfortunately, checking connectivity between merged runs is not as easy as it is for simple runs. See Fig. 5.4c and Fig. 5.4d as an example. The input image has five rows, which result in three merged rows. Merged runs enclosed in the red rectangle of Fig. 5.4d seem to be connected, but the corresponding connected components are not. Therefore, we need a special method for checking connectivity between merged runs, which must take into account the bits of the original image.

Let $\mathscr{R}(r, c_s, c_e) = [\![r, r+1]\!] \times [\![c_s, c_e]\!]$ be the merged run spanning rows

**Algorithm 6** Junction matrix filling algorithm used by BMRS. Parameters *junc* and *bits* are pointers to junction matrix data and image data; $h$ and $w$ are the image dimensions.

```
1:  procedure FillJunctionMatrix(junc, bits, h, w)
2:      h_merge ← (h + 1)/2
3:      data_width ← (w + 63)/64
4:      for i ∈ [[0, h_merge − 1]] do
5:          bits_u ← bits + data_width ∗ (2 ∗ i + 1)
6:          bits_d ← bits + data_width ∗ (2 ∗ i + 2)
7:          bits_dest ← junc + data_witdh ∗ i

8:          u_0 ← bits_u[0]
9:          d_0 ← bits_d[0]
10:         bits_dest[0] ← (u_0 ∨ (u_0 ≪ 1)) ∧
                           (d_0 ∨ (d_0 ≪ 1))
11:         for j ∈ [[1, data_width]] do
12:             u ← bits_u[j]
13:             u_shl ← u ≪ 1
14:             d ← bits_d[j]
15:             d_shl ← d ≪ 1
16:             if bits_u[j − 1] ∧ (1 ≪ 63) then
17:                 u_shl ← u_shl ∨ 1
18:             if bits_d[j − 1] ∧ (1 ≪ 63) then
19:                 d_shl ← d_shl ∨ 1
20:             bits_dest[j] ← (u ∨ u_shl) ∧ (d ∨ d_shl)
```

$r$ and $r+1$ and columns from $c_s$ to $c_e$. Two runs $P = \mathscr{R}(r, s, e)$ and $\Sigma = \mathscr{R}(r-2, h, t)$ must be checked for connectivity if $h \leq e+1$ and $t \geq s-1$. If these conditions are verified, the overlapping segment is delimited between $a = max(s, h) - 1$ and $b = min(e, t) + 1$. Let $u$ and $d$ be respectively the segments of row $r-1$ and $r$ between $a$ and $b$:

$$u = \{r-1\} \times [\![a, b]\!] \tag{5.3}$$

$$d = \{r\} \times [\![a, b]\!] \tag{5.4}$$

As confirmed by the examples in Fig. 5.5, runs $P$ and $\Sigma$ are connected iff the following operation gives a non-zero result $f$, called *junction flag*:

$$f = (u \vee (u \ll 1)) \wedge (d \vee (d \ll 1))$$

**Algorithm 7** Inline function used in BMRS for checking if two merged runs are connected, after having pre-calculated the junction matrix. Parameter $flag\_bits$ points to the junction data between the two lines, $s$ and $e$ mark the starting and ending coordinates of merged runs overlapping.

```
1: function IsConnected(flag_bits, s, e)
2:     s_base ← s/64
3:     s_bits ← s mod 64
4:     if s = e then
5:         return flag_bits[s_base] ∧ (1 ≪ s_bits)
6:     e_base ← (e + 1)/64
7:     e_bits ← (e + 1) mod 64
8:     if s_base = e_base then
9:         cutter ← (¬0LL ≪ s_bits) ⊕ (¬0LL ≪ e_bits)
10:        return flag_bits[s_base] ∧ cutter
11:    for i ∈ [[s_base + 1, e_base − 1]] do
12:        if flag_bits[i] then
13:            return true
14:    cutter_s ← ¬0LL ≪ s_bits
15:    cutter_e ← ¬(¬0LL ≪ e_bits)
16:    if flag_bits[s_base] ∧ cutter_s then
17:        return true
18:    if flag_bits[e_base] ∧ cutter_e then
19:        return true
20:    return false
```

Computing junction flags for each pair of runs that may be connected is expensive; therefore, junction flags are pre-calculated for all whole row pairs, and stored in the *junction matrix*, which occupies as much data as the merged input. The algorithm for building the junction matrix is described in Algorithm 6. Then, Algorithm 7 details how junction flags can be used to determine whether two runs are connected; it basically takes the segment of the junction matrix corresponding to their intersection and checks whether it is non-zero.

The first scan of BMRS is very similar to that of BRTS, but the input data considered by the raster scan consist of merged rows, instead of the original bitonal image. Connectivity between the current merged run and merged runs in the previous row is checked with the method just described.

The second scan of BMRS is more complex than that of BRTS.

In fact, the label of each merged run $P$ must be assigned to all of its foreground pixels, and this requires reading the value of all pixels in the rectangle covered by $P$, in order to find the foreground.

The memory requirements for both the proposed algorithms are summarized in Table 5.1.

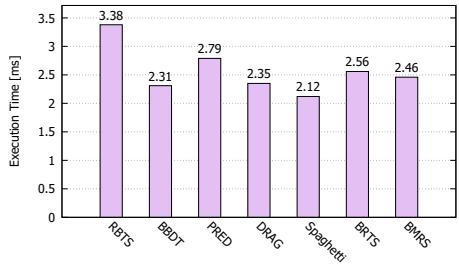



Figure 5.6: Average run-time tests on the *Medical* dataset. The input image is 1-byte-per-pixel for all the algorithms: times of BRTS and BMRS include the conversion to 1-bit-per-pixel format. Measured on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. All algorithms employ the UFPC label solver.

### 5.2.3 Experimental Results

The performance of the proposed algorithms has been evaluated using YACCLAB. YACCLAB, as already mentioned in Section 3.4, contains a large suite of real world datasets, covering most fields where CCL is usually employed. Experimental results discussed in the following were obtained on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. All the algorithms have been compiled for x64 architecture with optimizations enabled. YACCLAB includes an implementation of all the main CCL algorithms published in recent literature, for comparison with new proposals.

BRTS and BMRS have been compared to RBTS, BBDT, PRED, DRAG and Spaghetti, all introduced in Chapter 3. Among those, Spaghetti is the current state of the art. For a fair comparison, each algorithm is provided with input data in its preferred format, i.e., 1-bit-per-pixel for BRTS and BMRS, and 1-byte-per-pixel for all the others.

The label solver used by all algorithms is UFPC, which achieves the best performance on average [79]. Fig. 5.7 reports bar charts of average execution times on real world datasets.

Because all the compared algorithms employ a two-scan approach, time

measures are divided into memory allocation/deallocation, first scan and second scan, for a more fine-grained comparison.

Table 5.1: Memory requirement of BRTS and BMRS, compared to that of Spaghetti. Values are expressed in bytes per pixel, so the total amount can be obtained multiplying the values by $W \times H$.

| | Spaghetti | BRTS | BMRS |
|---|---|---|---|
| *Output Image* | 4 | 4 | 4 |
| *Label Solver* | 1 | 1 | 1 |
| *Run Metadata* | - | 4 | 2 |
| *Merged Input* | - | - | 1/16 |
| *Junction Matrix* | - | - | 1/16 |
| ***Total*** | 5 | 9 | 7+1/8 |

Table 5.2, instead, reports average total times and standard deviations. Some datasets, such as *Tobacco800*, contain images with a significant difference in resolution, causing high standard deviation in CCL performance. In all datasets, RBTS is by far the slowest algorithm; BBDT, PRED, DRAG and Spaghetti have similar performance, with Spaghetti being always the best of the four; BRTS and BMRS outperform all the competitors, and BMRS is always the fastest. In particular, the speedup of BRTS and BMRS wrt Spaghetti ranges from 1.16 to 1.34 and from 1.18 to 1.60 respectively. The overall improvement is greater on datasets of small images (Fingerprints and Mirflickr), where alloc/dealloc has the lowest impact on total execution time. Comparing the original RBTS algorithm to BRTS, it is clear that the first scan is the most optimized step, with an average speedup of 9.3. This huge performance improvement is due to the introduction of the FFS operation, which is available as a hardware instruction on 64-bit words on the selected test system, as long as in most modern CPU ISAs. The same optimization also make BRTS faster than state-of-the-art algorithms, which do not employ specific hardware operations or compiler intrinsics. The other proposal, BMRS, furtherly optimizes the first scan, approximately halving the amount of runs to be scanned and processed. Moreover, in spite of using two more data structures, the total memory needed by BMRS is actually less than that required by BRTS, allowing a little alloc/dealloc time saving.

Both the two proposals require more data structures than state-of-the-art algorithms: the total amount of allocated memory is, for BRTS, almost twice that of Spaghetti, as can be seen in Table 5.1. However, the extra memory needed is only equal to that of the output, and therefore does not represent an issue on most systems. Moreover, as demonstrated by the

experiments, the impact of these additional allocations on execution time is minimal.

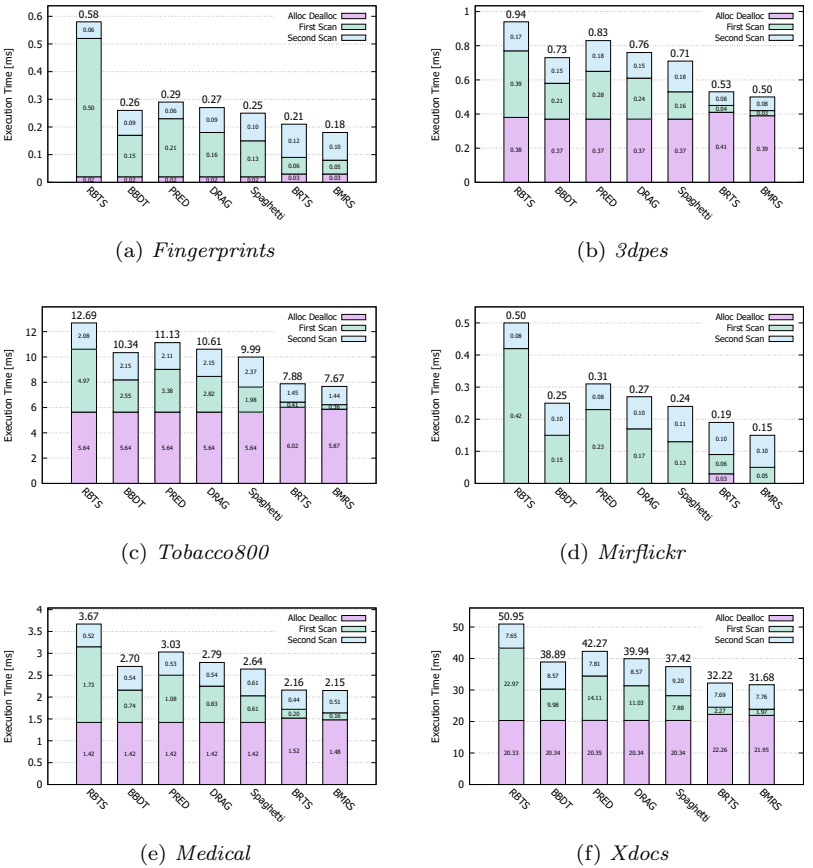
Another possible downside of the new proposals is their preferred bitonal input format. This can raise issues when working with libraries which do not natively support it, such as OpenCV as of today (release 4.5.2). If the input provided to BRTS or BMRS is not bitonal, both algorithms must begin with a conversion. Fig. 5.6 depicts average execution times on the Medical dataset in this case: the input is 1-byte-per-pixel for all the algorithms. As can be observed, BRTS and BMRS perform worse than current state of the art. However, the bitonal format is the most memory-efficient representation of binary images, and as such there are also some computer vision libraries that natively support it. An example is Leptonica, an open-source image processing library employed in several projects (e.g. tesseract OCR by Google). When the input is in bitonal format, BRTS and especially BMRS are the CCL algorithms of choice.

### 5.2.4 Conclusion

Two new run-based CCL algorithms have been presented, which employ Find First Set instructions to efficiently retrieve runs from a bitonal input. Experiments conducted over a collection of real world datasets demonstrate that both proposals outperform the fastest CCL algorithms in literature, thus representing the new state of the art for connecting components labeling on bitonal images. The source code is available in YACCLAB [78].

Table 5.2: Average run-time tests in ms, ± standard deviation. Results have been obtained on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. All algorithms employ the UFPC label solver. Novel proposals are marked with a star.

| | RBTS | BBDT | PRED | DRAG |
|---|---|---|---|---|
| *Fingerprints* | $0.58 \pm 0.20$ | $0.26 \pm 0.13$ | $0.29 \pm 0.14$ | $0.27 \pm 0.13$ |
| *3dpes* | $0.94 \pm 0.17$ | $0.73 \pm 0.05$ | $0.83 \pm 0.06$ | $0.76 \pm 0.06$ |
| *Tobacco800* | $12.69 \pm 7.78$ | $10.34 \pm 6.42$ | $11.13 \pm 6.90$ | $10.61 \pm 6.60$ |
| *Mirflickr* | $0.50 \pm 0.26$ | $0.25 \pm 0.80$ | $0.31 \pm 0.10$ | $0.27 \pm 0.08$ |
| *Medical* | $3.67 \pm 1.54$ | $2.70 \pm 1.14$ | $3.03 \pm 1.27$ | $2.79 \pm 1.18$ |
| *Xdocs* | $50.95 \pm 5.39$ | $38.89 \pm 4.26$ | $42.27 \pm 4.66$ | $39.94 \pm 4.37$ |
| | Spaghetti | BRTS* | BMRS* | |
| *Fingerprints* | $0.25 \pm 0.12$ | $0.21 \pm 0.11$ | $\mathbf{0.18} \pm 0.09$ | |
| *3dpes* | $0.71 \pm 0.05$ | $0.53 \pm 0.08$ | $\mathbf{0.50} \pm 0.06$ | |
| *Tobacco800* | $9.99 \pm 6.20$ | $7.88 \pm 4.91$ | $\mathbf{7.67} \pm 4.87$ | |
| *Mirflickr* | $0.24 \pm 0.08$ | $0.19 \pm 0.12$ | $\mathbf{0.15} \pm 0.06$ | |
| *Medical* | $2.64 \pm 1.12$ | $2.16 \pm 0.96$ | $\mathbf{2.15} \pm 0.95$ | |
| *Xdocs* | $37.42 \pm 4.11$ | $32.22 \pm 3.46$ | $\mathbf{31.68} \pm 3.70$ | |




(a) *Fingerprints* (b) *3dpes*

(c) *Tobacco800* (d) *Mirflickr*

(e) *Medical* (f) *Xdocs*

Figure 5.7: Average run-time tests with steps in ms (lower is better) on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. All algorithms employ the UFPC label solver.

# Chapter 6

# Connected Components Labeling on Massively Parallel Architectures

The fast advance of Graphic Processing Units (GPUs) of the last years encouraged the development of algorithms specifically designed to work in data parallel environments. Indeed, along with sequential solutions [61, 70, 121], many algorithms exploiting the parallelism of both CPUs and GPUs have been proposed [122, 123, 37, 86, 62]. Anyway, the computational throughput of GPUs tends to decrease when used for solving less regular problems: CCL GPU-based implementations tend to achieve comparable performance with respect to the CPU ones [86, 62]. Even when the execution time over CPU algorithms is lower, the improvement is unable to compensate time-consuming data transfer between host and device. At this point the reader would probably raise the question "why should we prefer GPU implementations instead of relying on classical well performing CPU algorithms?" It is soon said: the high data transfer cost means that all the applications that entirely execute on GPU would certainly benefit from an optimized GPU-based labeling algorithm, without the need of moving back and forth from CPU just to retrieve image objects.

Many of the improvements introduced for sequential implementation has been sooner or later ported to GPU algorithms, or inspired for addi-

tional optimizations on such architectures. As an example, the *union-find* data structure originally applied to CCL in [124] to solve equivalences between labels, and later employed by different authors [33] for implementing GPU algorithms. The *block-based* approach is another noticeable example; introduced in [29] and subsequently employed in different CPU-based proposals [57, 125, 102] has been ported to GPU-based algorithms [85].

This Chapter is about Connected Components Labeling (CCL) algorithms developed for GPU accelerators. The task itself is employed in many modern image processing pipelines and represents a fundamental step in different scenarios, whenever object recognition is required. For this reason, a strong effort in the development of many different proposals devoted to improve algorithm performance using different kind of hardware accelerators has been given.

The next Section provides a review of GPU-based algorithmic solutions published in the last two decades, highlighting their distinctive traits and the improvements they leverage on. The review is equipped with the source code, available in YACCLAB [78], which allows to straightforwardly reproduce all the algorithms in different experimental settings.

Section 6.2, Section 6.3 and Section 6.4 introduce novel algorithmic solutions, which establish the new state of the art for CCL on GPUs. Then, in Section 6.5, extensive tests are performed over multiple settings to evaluate all the algorithms, with a deep performance analysis and discussion. Finally, Section 6.6 draws conclusive remarks and future research directions.

## 6.1 Literature Survey

There are many different ways to categorize GPU CCL algorithms. Broadly speaking, one of the most common categorization distinguish between *iterative* and *direct* solutions. The former repeat one or more kernels a data-dependent amount of times. Usually, they iterate until no more changes on data are detected. These algorithms tend to tolerate race conditions, demanding the correction of wrong results to the next iteration. Direct algorithms, on the other hand, perform a fixed number of kernel executions. Differently from iterative algorithms, each kernel must produce an exact result: therefore, most direct algorithms employ hard concurrency control

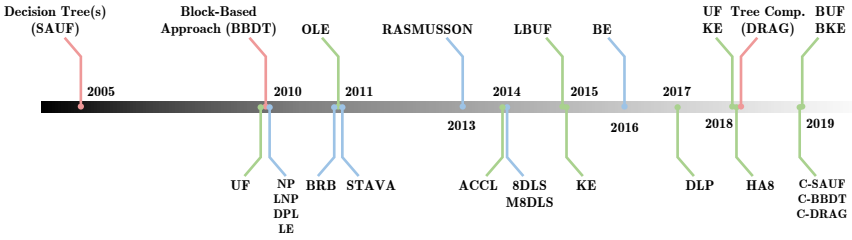



Figure 6.1: Timeline of algorithm publication. Red lines represent CPU algorithms that introduced breakthrough changes in the labeling procedures, later included in GPU implementations. Blue and green lines are respectively used to identify iterative and direct solutions. Few algorithms appear twice in the timeline, meaning that the algorithm was initially released for a single connectivity type (e.g. 4-connectivity), and later extended to work with other types of connectivity (e.g. 8-connectivity) or to 3D volumes.

to avoid race conditions, usually in the form of atomic operations.

Another possible classification may be based on the minimal set of pixels (or voxels) that are processed together. In this sense, pixel-based, run-based, and block-based algorithms can be identified:

- Pixel-based algorithms are the simplest: each foreground pixel has its own label;
- Run-based algorithms work on *runs*, i.e., chunks of consecutive foreground pixels in the same row, and assign labels to these runs instead of single pixels. This approach is beneficial in some scenarios, but performance significantly drops when foreground objects within images have elongated shapes on the y-direction;
- Block-based algorithms, originally proposed in [29], label $2 \times 2$ blocks of pixels at once. This is possible because, when considering 8-connectivity, any two foreground pixels in a $2 \times 2$ block are connected to each other. When moving to 3D the concept of blocks can remain at slice level or be extended to the third dimension, considering $2 \times 2 \times 2$ blocks.

In the following of this Section we will describe state-of-the-art solutions, distinguishing between iterative algorithms and direct ones.

**Algorithm 8** Neighbour Propagation and Local Neighbour Propagation kernels. $I$ is the input image, $L$ is the output label image, *id* is the thread identifier, corresponding to a pixel raster index.

```
1: kernel NP(I, L, r, c)
2:     l ← L(r, c)
3:     for all (n_r, n_c) ∈ N(r, c) do
4:         if I(n_r, n_c) = I(r, c) ∧ L(n_r, n_c) < l then
5:             l ← L(n_r, n_c)
6:             changed ← true
7:     L(r, c) ← l
8: kernel LNP(I, L, r, c)
9:     l ← L(r, c)
10:    for all (n_r, n_c) ∈ N(r, c) do
11:        if I(n_r, n_c) = I(r, c) ∧ L(n_r, n_c) < l then
12:            l ← L(n_r, n_c)
13:            changed ← true
14:    __shared__ sL[blockDim.x × blockDim.y]
15:    sChanged ← false
16:    (r_s, c_s) ← (threadIdx.y, threadIdx.x)
17:    loop
18:        sL(r_s, c_s) ← l
19:        __syncthreads()
20:        sChanged ← false
21:        for all (n_r, n_c) ∈ N(r, c) do
22:            if I(n_r, n_c) = I(r, c) then
23:                if sL(n_r, n_c) < l then
24:                    l ← sL(n_r, n_c)
25:                    sChanged ← true
26:        __syncthreads()
27:        if sChanged = false then
28:            break
29:    L(r, c) ← l
```

The timeline reported in Fig. 6.1 depicts temporal release of each algorithm, providing the reference context.

Later in this section, a table summarizing all the characteristics of the algorithms (i.e., publication year, authors, minimal set of pixels analyzed, scanning procedure, connectivity and input supported, and data structure(s) required) is also provided (Tab. 6.1).

### 6.1.1 Iterative Algorithms

#### Neighbor Propagation (NP)

The first work that addresses the GPU CCL problem is due to Hawick *et al.*, and is dated back to 2010 [126]. The authors deal with the general problem of identifying CCs in graphs, and then propose four specific algorithms for d-dimensional hypercubic meshes, which include binary images. The first, and simplest, of these proposals is Neighbor Propagation. All foreground pixels are initialized with sequential values, each equal to the pixel raster index. Then, the provisional label of each pixel is updated to the minimum of its neighbor labels. This operation must be repeated until convergence, i.e., until there are no more changes in the output image. For this reason, the algorithm is inserted in the

iterative group. The pseudocode for this algorithm is provided in Alg. 8.

### Local Neighbor Propagation (LNP)

Local Neighbor Propagation is an optimization of NP that initializes the output image, splits it into blocks (16x16 in the original paper) and performs the update operations described above on each block separately. The advantage over the previous solution is that, in this case, the shared memory inside the block is exploited. As NP, each thread updates a different pixel. The update kernel is divided into two steps: during the first step, each thread loads into shared memory the minimum neighbor label of its pixel, reading from device memory; then, the second step updates each label with the minimum of the neighbors, same as NP, but this time performed locally, in shared memory. This last operation is repeated until block convergence. Then, the content of the shared memory is copied into device memory, and the procedure restarts from the beginning, until global convergence is reached. Alg. 8 details the kernels of the algorithm.

### Directional Propagation Labeling (DPL)

Directional Propagation Labeling (DPL), again proposed in the same paper as NP and LNP [126], tries to overcome the limit of the neighborhood-confined propagation. In order to accomplish this, it assigns to each thread the task of propagating the minimum label through a row or a column. This operation is performed alternatively on four directions: left to right, bottom to top, right to left and top to bottom. Unlike the two previous algorithms, this strategy is only compatible with 4-connectivity.

### Label Equivalence (LE)

Label Equivalence is the last and certainly the most interesting proposal of the paper from Hawick *et al.* [126]. It is also one of the first iterative algorithms to record and solve equivalences using an auxiliary data structure.

The paper does not explicitly refer to the union-find, but the operations performed are the same and can be described by referring to them. In the first kernel, `Init`, the output image $L$ is initialized as usual, with each pixels being given a provisional label equal to its raster index. Moreover, an additional data structure with the same size of the output image, $H$, is

allocated and initialized in the same way. The algorithm then consists of three more kernels that are repeated in sequence until convergence: `Scan`, `Analysis`, and `Labeling`. The first kernel, `Scan`, is responsible for updating equivalences between pixels. Each thread works on a different pixel, finding the smallest of the neighbor labels and updating the corresponding value in $H$ with the minimum label found. This operation does not touch $L$, and only modifies $H$. The second kernel, `Analysis`, performs a compression of the equivalence tree, i.e., each thread is responsible for replacing a pixel of $H$ with the root of the tree. This operation corresponds to *find*. The third and last kernel, `Labeling`, updates the labels in the output image $L$ with the values contained in $H$. During Scan kernel, the algorithm requires atomic operations to avoid lost updates on $H$, in the case that two threads happen to concurrently update the same pixel.

After its appearance, LE has been improved by many authors. Kalentev*et al.* [127] noticed that the need for a separate data structure to store label equivalences could be removed through the direct use of the output image $L$. In this optimization only two kernels are iterated after the initialization. `Analysis` kernel performs the compression of trees directly on the output image, updating the temporary label of each pixel. Thus, there is no need for a separate kernel devoted to copy the labels. In `Scan` kernel, every thread finds the smallest neighbor label, $nl$, and assigns it to the father (with label $fl$) iff $nl < fl$. Kalentev *et al.* also removed the atomic operations simply increasing the number of iterations. The authors claim that atomic operations slow down computation more than additional iterations caused by collisions do. Kernels of this optimized version of Label Equivalence are detailed in Alg. 9.

### Stava and Benes (STAVA)

Stava and Benes [128] proposed a solution that can be seen as an improvement of LE, obtained with the use of Tiles Merging. The algorithm is composed of four kernels: `LocalLabeling`, `GlobalLabeling`, `BorderCompression` and `PathCompression`. In the first kernel, `LocalLabeling`, $16 \times 16$ tiles are labeled with a light version of LE, which does not require any additional union-find structure to store label equivalences, similar to the one proposed by Kalentev. This kernel is detailed in Alg. 12.

The second kernel, `GlobalMerge`, merges the labeled tiles, in a hierarchical order: in step $n$, macro-tiles are obtained by merging groups of

**Algorithm 9** Label Equivalence kernels. $I$ is the input image, $L$ is both union-find array and output label image, $id$ is the thread identifier, corresponding to a pixel raster index

```
 1: kernel INIT(I, L, r, c)
 2:     if I(r, c) = 1 then
 3:         L(r, c) ← r * w + c + 1
 4:     else
 5:         L(r, c) ← 0

 6: kernel SCAN(L, r, c, changes)
 7:     l ← L(r, c)
 8:     if l > 0 then
 9:         min_l ← FindMinNeighborLabel(r, c)
10:         if min_l < l then
11:             L[l − 1] ← min(L[l − 1], min_l)
12:             changes ← true

13: kernel ANALYSIS(L, r, c)
14:     l ← L(r, c)
15:     if l > 0 then
16:         L(r, c) ← Find(L, l)
```

$4 \times 4$ tiles, output of step $n - 1$. Each group of tiles is processed by a three-dimensional thread block, where $x$ and $y$ components of the thread-id locally identify a tile. A *union* is performed between each pixel of the South and East border and its neighbors outside the tile. For each tile, the merging process is repeated until no more changes are detected. For optimization purpose, `GlobalMerge` is alternated to `BorderCompression`, which performs *find* on border pixels only: the aim is to shorten union-find trees and consequently speed-up the next round of global merge.

Finally, the last kernel, `PathCompression`, performs a *find* on all pixels of the output image, thus completing the CCL process.

### Block-Run-Based (BRB)

Block-Run-Based connected components labeling (BRB) [129], as the name suggests, combines block-based and run-based strategies in order to simplify the equivalence label solving process. The whole algorithm is im-

plemented in GPU shared memory, thus minimizing global memory bandwidth consumption. The image, of width $W$ (and half-width $HW$), is divided into overlapping Block Rows (BR), composed of two consecutive pixel rows, where the upper pixel row is shared with the previous block row. Each thread works on a $2 \times 2$ block. The algorithm is composed of two scans, and the first scan is divided into three steps: `BlockRunExtraction`, `PreviousBRUpdate` and `CurrentBRUpdate`.

In the first step, the current BR is compressed into one pixel row by applying OR operation of its upper and lower row pixel value. Then, a run of contiguous 1s is detected from the compressed pixel row and the column index of the block containing the run starting pixel is assigned to each block in this run, by means of `__ballot_sync`[1] and `__clz` (Count Leading Zeros) instructions. From another perspective, each block can be considered as a node of a union-find tree, while the label value represents a linkage address pointing to the column location of its parent node. The purpose of the following two iterative steps is to merge union-find trees of current and previous BRs according to the connection relations. The second step, `PreviousBRUpdate`, merges union-find trees of pairs of connected blocks in the current and previous row, by linking one root to the other one. Then, `CurrentBRUpdate` is similar to the previous step, but only updates root nodes in the current BR. These two steps are iterated until convergence.

The second scan, `LabelAssignment`, proceeds row-wise in reversed direction, and assigns a unique global label to each block recognized as a tree root. Then, child blocks get their label from the parent node.

Unfortunately, the implementation available only deals with specific image dimensions (e.g. $512 \times 512$ and $1024 \times 1024$). Any different size would require a specific implementation or multiple conditional checks that would significantly degrade the performance.

### 8-Directional Label Selection (8DLS)

8-Directional Label Selection [130] is another iterative algorithm in which, at every step, each pixel is assigned the minimum label found scanning each of the 8 directions radiating from it until a background pixel is encountered.

[1] `__ballot_sync` is a warp vote function: it evaluates a predicate for all the threads in the warp and returns an integer whose n-th bit is set iff the predicate evaluates to non-zero for the n-th thread.

Modified 8-Directional Label Solver (M8DLS) is an improved version of 8DLS, that avoids processing pixels which already have minimum label. After two iteration of 8DLS, pixels that still have their original label are considered permanent, and stop being updated. The specific amount of iterations, 2, is empirically determined, based on exhaustive tests conducted by the authors. Unfortunately, when using such a kind of "optimization" the output correctness is not ensured, making the algorithm not always correct.

### Rasmusson

Rasmusson *et al.* [131] proposed a method based on early calculation of label propagation sizes, from the connectivity between pixels extracted in a pre-processing step, and reuse of established label propagation routes. As usual, the algorithm starts by initializing the output with provisional labels equal to pixel raster indices. However, the four most significant bits of each label store connection information about half the neighborhood; because of this trick, the algorithm only works on images with at most $2^{28} \approx 256$M pixels.

The core of the algorithm is the `Propagate` kernel, which is repeated until convergence. The image is divided into $32 \times 32$ tiles, which are copied into shared memory, and a thread is created for each pixel. Then, for each of the 8 possible directions starting from a pixel, the maximum propagation is computed. These propagation sizes are iteratively calculated in a parallel manner: each thread stores the provisional propagation size in a local variable, $p$, and increments it at each step by the value of the provisional propagation size of the pixel distant $p$ positions. In this way, a maximum propagation of 32 is reached after $\log_2 32 = 5$ iterations. Different threads share the respective values of $p$ by means of shuffle operations, which efficiently exchange a variable between threads within a warp. The pixel-thread association must change for each direction, in order for pixels aligned in a certain orientation to always be in the same warp. The label of each pixel is updated to the minimum of the 8 labels in the maximum propagation sizes, and also a propagation size of 1 is considered for the correctness of corner cases. In order to extend the label propagation to neighbor tiles, actually $33 \times 33$ blocks are copied into shared memory.

#### Block Equivalence (BE)

Proposed by Zavalishin *et al.* [85], Block Equivalence (BE in short) is a block-based version of Label Equivalence (Sec. 6.1.1), enriched with a novel method for checking block connectivity in a parallel fashion. The algorithm is composed of four kernels: `Init`, `Scan`, `Analysis`, and `FinalLabeling`, each requiring a thread per block. The `Init` kernel is responsible for finding which pairs of neighbor blocks are indeed connected. Decision trees, used by the original block-based CCL algorithm [29], tend to cause thread divergence and inefficiency in a GPU environment; therefore, a different method is proposed, that avoids nested conditional jumps.

1. Each internal pixel of block $X$ is read, in order to find out which neighbor blocks could possibly be connected to $X$;

2. Internal pixels of the neighbor blocks selected by the previous step are read, and the hypothesis of connectivity with $X$ is confirmed or refused.

At the end of `Init`, block adjacency information is stored in an ad-hoc matrix $BA$, to be read again in the subsequent phases. Each block has 8 adjacent blocks, so one bitmapped byte per block is sufficient to record the neighbors (adjacent blocks sharing the same value). Kernels `Scan` and `Analysis` behave in the same way as the kernels of Label Equivalence, in the optimized version of Kalentev *et al.*; the only difference is that they work on blocks instead of pixels, storing block labels in a specific matrix, $BL$. `Scan` and `Analysis` are iterated until convergence, and then `FinalLabeling` copies block labels into pixel labels, thus composing the output. The pseudo-code is provided in Alg. 10.

### 6.1.2 Direct Algorithms

#### Union Find (UF)

Union Find, by Oliveira and Lotufo [33], is chronologically the first non-iterative proposal, and also one of the first GPU CCL algorithms overall. It consists of a parallel version of the common union-find algorithm, largely used in sequential CCL solutions. UF is based on three kernels: `Initialization`, `Merge` and `Compression`. Each kernel is launched on a number of threads equal to the image size, and each thread is assigned a

**Algorithm 10** Block Equivalence `Init` kernel. $I$ is the input image, $L$ is both union-find array and output label image, $id$ is the thread identifier, corresponding to a pixel raster index.

```
1:  kernel INIT(I, bL, bConn, r, c)
2:      P ← 0
3:      P0 ← 777_16
4:      if I[2r, 2c] > 0 then
5:          P ← P | P0
6:      if I[2r, 2c+1] > 0 then
7:          P ← P | (P0 ≪ 1)
8:      if I[2r+1, 2c] > 0 then
9:          P ← P | (P0 ≪ 4)
10:     if I[2r+1, 2c+1] > 0 then
11:         P ← P | (P0 ≪ 5)
12:     if P > 0 then
13:         bL[r, c] ← r∗w+c+1
14:         if HasBit(P, 0) ∧ I[2r−1, 2c−1] > 0 then
15:             SetBit(bConn[r, c], 0)
16:         if (HasBit(P, 1) ∧ I[2r−1, 2c] > 0) ∨
17:           (HasBit(P, 2) ∧ I[2r−1, 2c+1] > 0) then
18:             SetBit(bConn[r, c], 1)
19:         ...
```

pixel, $x$. In order to save avoidable and time consuming memory allocations, no additional memory is reserved for the union-find data structure. Instead, the equivalences between labels are stored, during the procedure, in the output image itself, in the sense that the provisional label assigned to a pixel corresponds to the linear index of its father node in the union-find forest. In the `Initialization` kernel, all foreground pixels in the output image are initialized with a provisional label equal to their linear index. From the union-find point of view, this procedure corresponds to the creation of a separate tree for every pixel. In kernel `Merge`, each thread working on a foreground pixel $x$ checks its neighborhood mask, and for every foreground neighbour it performs a *union* with $x$. Since *union* is a symmetric operation, the neighborhood mask only contains half of the neighbors of a pixel. The *union* operation can possibly involve reading and writing operations on other pixels than $x$. The possibility that two or

**Algorithm 11** `LocalMerge` kernel of UF (Sec. 6.1.2). $I$ is input image, $L$ is both union-find array and output label image, $id$ is the thread identifier, corresponding to a pixel raster index.

```
1: kernel LOCAL MERGE(I, L, r, c)
2:     __shared__ sI[]
3:     __shared__ sL[]
4:     r_s ← threadIdx.y
5:     c_s ← threadIdx.x
6:     sI(r_s, c_s) ← L(r, c)
7:     __syncthreads()
8:     for all (n_r, n_c) ∈ N(r, c) do
9:         if I(r, c) = I(n_r, n_c) then
10:            Union(sL, id_s, id_sn)
11:    l ← Find(sL, id_s)
12:    L[id] ← convertLabelToGlobal(l)
```

more threads read or modify the same pixel is taken into account through the use of atomic operations in *union*, in order not to lose updates, and avoid the necessity of multiple iterations. After `Merge`, every connection between pixels in the image is reflected in the union-find structure, in the sense that every separate tree represents a connected component. Finally, the `Compression` kernel performs the compression of trees. This operation makes sure that every pixel is assigned a label corresponding to the linear index of the root of its tree. Thus, every pixel in the same CC ends up sharing the same label, and the labeling task is completed. The aforementioned basic structure of the algorithm is enhanced with the addition of another common technique known as Tile Merging. This means that the entire algorithm is first performed on rectangular blocks the image is divided into in a preliminary phase called `Local Merge`. This allows to take full advantage of shared memory. This local phase is detailed in Alg. 11. Then, the `Merge` kernel is performed on border pixels only, and a final `Compression` is executed over the entire image.

### Line-Based Union Find (LBUF)

Line-Based Union Find [34] is a variation of UF that employs single lines as tiles for the Tile Merging strategy. This choice reduces the neighbor-

hood of any pixel to a subset containing only the two neighbors belonging to the same row, and consequently allows to simplify the logic of `Local Merge`. In fact, because only half the neighborhood needs to be checked (see Sec. 6.1.2), each thread only has to link its pixel to the one on the left, in the case that both are foreground. The remaining kernels of UF are left unchanged.

### Komura Equivalence (KE)

Komura Equivalence [35] can be seen as a variation of UF, which involves a more complex initialization in order to remove the need for one Union per pixel later. It consists of four steps: `Initialization`, `Compression`, `Reduction`, and `Compression` again. The main difference between KE an UF lies in the first kernel: differently from UF, KE `Initialization` does not create single-node trees. Instead, every thread checks the neighborhood of $x$ in increasing raster index order, and the smallest foreground neighbor is set as the father node of $x$. This first phase aims at creating partial union-find trees right from the beginning, that are flattened by the subsequent `Compression` kernel. The `Reduction` kernel is a variation of `Merge`. It only performs a *union* between $x$ and foreground neighbors that were not chosen during `Initialization`. Analogously to UF, a final `Compression` is required for flattening the forest trees.

### Accelerated CCL (ACCL)

The CUDA compute capability 3.5 introduced several features, among which dynamic parallelism and Hyper-Q are the ones exploited by the Accelerated CCL (ACCL) algorithm, proposed by Paravecino and Kaeli in [132]. ACCL configures as a run-based algorithm, and is composed of two phases. In the first phase, a thread is run for each row of the image, with the aim of finding runs and recording first and last indices. During this step, each run is assigned a provisional label, recorded in a separate matrix, L. During the second scan, connected runs from contiguous rows are joined. This particular operation involves spawning a child kernel for each merging, with one thread per provisional label in L. These threads update the respective labels of the added segment. The algorithm makes use of texture memory for read-only memory accesses, which improves the performance when multiple threads from the same block access contiguous

memory locations. Finally, the Hyper-Q feature is exploited to process multiple images at the same time with different CUDA streams, thus increasing the total throughput.

### Distanceless Label Propagation (DLP)

Proposed by Cabaret *et al.* [37], Distanceless Label Propagation is another algorithm based on union-find, characterized by an uncommon gather-scatter procedure used to propagate the minimum label of a CC. It includes four kernels: `DLP-I`, `DLP-SR`, `DLP-R` and `DLP-RUF`. The first kernel, `DLP-I`, initializes foreground pixels as usual, with a provisional label equal to the raster index. The second kernel is `DLP-SR` (SetRoot), and is responsible for a gather-scatter label propagation procedure. The minimum label is found in a mask selecting a reduced neighborhood of $x$, containing $2 \times 2$ pixels ($x$, right, down, and down-right); then, differently from most algorithms based on minimum label gathering, the minimum is scattered back to the whole mask with the SetRoot procedure: for each involved pixel, the root of its union-find tree is updated with the minimum label just found, by means of the usual atomic operation that characterizes direct algorithms. The third kernel, `DLP-R`, performs the classical compression of union-find trees, replacing provisional labels with the root of their trees. The last kernel is `DLP-RUF`, named after its distinctive algorithm, *recursive* union-find. It is the same as `DLP-SR`, but replaces the SetRoot procedure with a recursive implementation of union. These four different kernels are used to build three sequential steps, which compose the CCL algorithm:

1. Tile Labeling: the labeling process is performed locally on tiles, by running in sequence `DLP-I`, `DLP-SR`, `DLP-R`, `DLP-RUF` and finally another `DLP-R`;
2. Border Merging: `DLP-RUF` is applied to border pixels only, with the aim of merging union-find trees of different tiles;
3. Relabeling: `DLP-R` is applied to the whole image.

### Hardware Accelerated 4/8 (HA4/HA8)

In 2018, Hennequin*et al.* [133] proposed HA4, a 4-connected run-based algorithm heavily based on CUDA intrinsics. The algorithm can be used to perform either connected components labeling or analysis; the CCL

variation is composed of three kernels: `StripLabeling`, `BorderMerging` and `FinalLabeling`.

The first kernel, which is also the most complex, performs a partial labeling on horizontal strips. The block width is 32, same as the warp size, and each threads is assigned a pixel $X$, so that a line of 32 pixels perfectly matches a warp. The kernel begins with threads reading the values of their pixels and sharing them to the rest of the line (warp) with a `__ballot_sync` instruction, which produces a 32-bit bitmask of the foreground pixels; then, each thread checks the *start* and *end* position of the run $X$ belongs to, by means of efficient `__ffs` (Find First Set) and `__clz` intrinsics. The first pixel of each run is given a temporary label equal to its raster index, while the other pixels are left uninitialized. Then, in order to merge lines vertically, the first thread of the line copies the bitmask generated earlier with `__ballot_sync` in shared memory, so that it can be read from the lower line. Each thread performs a *union* if $X$ or the pixel above is the start of a run. After this partial labeling of the block, the same threads shift 32 pixels to the right, and continue labeling the stripe with the same method, extending runs that overcome the block border. This technique avoids the need for merging vertical borders, and minimizes thread creation/destruction.

The second kernel merges the horizontal border, performing a *union* only on the starts of runs, same as in `StripLabeling`. Finally, the last kernel performs the final labeling. Only the run-starting-pixel thread retrieves the final label with a *find*, and then it propagates it to the other threads with a shuffle operation; finally, each thread updates the label image with this value.

Since the original algorithm is 4-connected, we propose a 8-connected variation, which performance can be compared to the other algorithms of this survey. In order to make the algorithm 8-connected, it is sufficient to add the diagonal direction to the horizontal border merging, and perform a *union* if the rightmost of the two adjacent pixels is a start. Specifically, each thread must merge $X$ to the upper-left pixel if $X$ is a start, and to the upper-right pixel if the latter is a start.

Checking and merging in diagonal direction gets tricky at warp borders, since the diagonal neighbor pixels can fall into different warps. To tackle such cases, blocks of the `BorderMerging` kernel encompass 1024 pixel-long lines, where each warp shares the value of the rightmost pixel to the subsequent warp using shared memory.

**Algorithm 12** Stava and Benes `LocalMerge` kernel. $I$ is the input image, $L$ is both union-find array and output label image, $id$ is the thread identifier, corresponding to a pixel raster index.

```
1:  kernel LocalMerge(I, L, r, c)
2:      __shared__ sI[16 × 16]
3:      __shared__ sL[16 × 16]
4:      __shared__ changes[1]
5:      loc_r ← threadIdx.y
6:      loc_c ← threadIdx.x
7:      sI(loc_r, loc_c) ← I(r, c)                          ▷ Load input patch
8:      __syncthreads()
9:      l ← loc_r ∗ blockDim.x + loc_c
10:     loop                                          ▷ Pass 1 of the CCL algorithm
11:         sL(loc_r, loc_c) ← l
12:         if threadIdx.x = 0 ∧ threadIdx.y = 0 then
13:             changes[0] ← false
14:         __syncthreads()
15:         new_l ← l                                 ▷ Find the minimal neighbor label
16:         for (n_r, n_c) ∈ 𝒩(r, c) do
17:             if sI(loc_r, loc_c) = sI(n_r, n_c) then
18:                 new_l ← min(new_l, sL(n_r, n_c))
19:         __syncthreads()
20:         if new_l < l then                         ▷ Merge the equivalence trees
21:             atomicMin(sL[l], new_l)
22:             changes[0] ← true
23:         __syncthreads()
24:         if changes[0] = false then
25:             break                                 ▷ Local solution has been found
26:         l ← Find(sL, l)                                              ▷ Pass 2
27:         __syncthreads()
28:     g_l ← convertLabelToGlobal(l)
29:     L(r, c) ← g_l                                 ▷ Store result to device memory
```

Table 6.1: Comparison between state-of-the-art algorithms, which considers publication year, authors, minimum block of pixels processed together (i.e, single pixel, block or run), processing type (i.e., iterative or direct), connectivity, input type (2D or 3D) and data structures employed in addition to the output image $L$. For each additional data structure its dimension in bytes is also specified in the form $width \times height \times bytes$. These structures have the same name used in the algorithm description. Some of the iterative algorithms require an additional byte (Termination Byte, TB) to check whether the process is completed or not after each iteration. Please note that bytes are specified considering 32-bit labels.

| *Name* | *Year* | *Authors* | *Block Size* | *Iterative* vs *Direct* | *Connectivity* | *Input* | *Additional Data Structure(s)* |
|---|---|---|---|---|---|---|---|
| NP [126] | 2010 | Hawick *et al.* | Pixel | Iterative | 4, 8 | 2D | |
| LNP [126] | 2010 | Hawick *et al.* | Pixel | Iterative | 4, 8 | 2D | |
| DPL [126] | 2010 | Hawick *et al.* | Run | Iterative | 4 | 2D | |
| LE [126] | 2010 | Hawick *et al.* | Pixel | Iterative | 4, 8 | 2D | $H = w \times h \times 4$ TB |
| UF [33] | 2010 | Oliveira and Lotufo | Pixel | Direct | 4 | 2D | |
| BRB [129] | 2011 | Chen *et al.* | Block & Run | Iterative | 8 | 2D | |
| OLE [127] | 2011 | Kalentev *et al.* | Pixel | Iterative | 8 | 2D | TB |
| STAVA [128] | 2011 | Stava and Benes | Pixel | Iterative | 8 | 2D | |
| RASMUSSON [131] | 2013 | Rasmusson *et al.* | Pixel | Iterative | 8 | 2D | |
| 8DLS [130] | 2014 | Soh *et al.* | Pixel | Iterative | 8 | 2D | TB |
| M8DLS [130] | 2014 | Soh *et al.* | Pixel | Iterative | 8 | 2D | |
| ACCL [132] | 2014 | Paravecino and Kaeli | Pixel | Direct | 4 | 2D | |
| LBUF [34] | 2015 | Yonehara and Aizawa | Pixel | Direct | 4, 8 | 2D | |
| KE [35] | 2015 | Komura | Pixel | Direct | 4 | 2D | |
| BE [85] | 2016 | Zavalishin *et al.* | Block | Iterative | 8 | 2D, 3D | $BL = \frac{w}{2} \times \frac{h}{2} \times 4$ $BA = \frac{w}{2} \times \frac{h}{2} \times 1$ TB |
| DLP [37] | 2017 | Cabaret *et al.* | Pixel | Direct | 8 | 2D | |
| HA8 [133] | 2018 | Hennequin *et al.* | Run | Direct | 4, 8 | 2D | |
| KE [36] | 2018 | Allegretti*et al.* | Pixel | Direct | 8 | 2D | |
| UF [36] | 2018 | Allegretti*et al.* | Pixel | Direct | 8 | 2D, 3D | |
| C_SAUF [58] | 2019 | Allegretti *et al.* | Pixel | Direct | 8 | 2D | |
| C_BBDT [58] | 2019 | Allegretti *et al.* | Block | Direct | 8 | 2D | |
| C_DRAG [58] | 2019 | Allegretti *et al.* | Block | Direct | 8 | 2D | |
| BUF [62] | 2019 | Allegretti *et al.* | Block | Direct | 8 | 2D, 3D | |
| BKE [62] | 2019 | Allegretti *et al.* | Block | Direct | 8 | 2D, 3D | |

## 6.2 Optimizing GPU Algorithms

As introduced in the previous Section, many approaches to compute GPU-CCL have been published in the last decade.

This Section aims to improve state-of-the-art GPU algorithms with two main contributions: *(i)* extension from 4- to 8-connectivity of UF and KE methods, and *(ii)* optimization of KE 8-connected.

For better describing the algorithmic solution introduced in this Section, we need to detail some of the algorithms already introduced in Section 6.1.

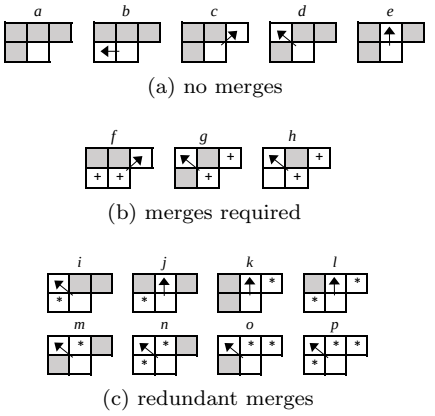



Figure 6.2: Possible neighbourhood configuration of a foreground pixel. Grey squares are background and white are foreground. Arrows show connections performed in `Initialization` phase, crosses identify mergers that must be performed during `Reduction` phase, stars highlight pixels to whom the merging operation can be delegated.

### Union-Find Algorithm

Oliveira *et al.* proposed in [33] a GPU-CCL algorithm based on a *union-find* approach. The *union-find* data structure was firstly applied to CCL problems by Dillencourt *et al.* in [30]. It consists of a forest of trees that supports two basic operations: *find* and *union*. The *find* function takes a node of a tree as input and returns its root as output. The *union* procedure takes two nodes as inputs and joins together the trees they belong to, by setting the first tree root as the father of the second one.

When union-find is applied to CCL, each pixel in the image matches a node in the data structure. The final goal is to build a single tree for each connected component, starting with every foreground pixel in its own tree and taking advantage of *union* for merging trees of connected pixels. The union-find forest can be stored in memory as an array, where each node is represented by an index, and the value stored at that index represents its parent node.

Root nodes have their own indexes stored in the array. Values in said array can also be interpreted as pixel labels.

This way, when the goal of matching every single connected component to a separate tree has been achieved, a flattening operation that links each node of every tree directly to the root is sufficient to assign the same label to every pixel in the tree, thus completing the CCL task. The union-find array can at this point be interpreted as the label image, saving the need for a separate data structure in memory. In order to distinguish between background and foreground pixels, in our implementation we actually write, for each node, the index of its parent + 1. This way, every foreground pixel has a positive label, and 0 is assigned to background ones.

**Algorithm 13** *Union-Find* procedures. $I$ is input image, $L$ is both *union-find* array and output label image.

```
 1: function FIND(L, index)
 2:     label ← L[index]
 3:     while label − 1 ≠ index do
 4:         index ← label − 1
 5:         label ← L[index]
 6:     return index

 7: procedure UNION(L, a, b)
 8:     done ← false
 9:     while done = false do
10:         a ← Find(L, a)
11:         b ← Find(L, b)
12:         done ← (a = b)
13:         if a = b then
14:             done ← true
15:         else
16:             if done ← false and a > b then
17:                 Swap(a, b)
18:             old ← atomicMin(&L[b], a + 1)
19:             done ← (old = b + 1)
20:             b ← old − 1
```

A possible implementation of *find* and *union* procedures is shown in Algorithm 13. Atomic operations are used in *union* to avoid problems concerning the simultaneous updates performed by different threads.

UF algorithm is based on the union-find data structure and it consists of three kernels: `Initialization`, `Merge` and `Analysis`, described in Algorithm 14. Each of them is launched on a number of threads equal to the image size, and each thread is assigned a pixel, which we will refer to as the *active pixel*.

During `Initialization` phase, the union-find structure is initialized, with each node in its own tree.

**Algorithm 14** Union-Find kernels. $I$ is input image, $L$ is both *union-find* array and output label image. Checks on image borders are not shown.

```
1: kernel Initialization(I, L, r, c)
2:     if I[r, c] = 1 then
3:         L[r, c] ← LinearIndex(r, c) + 1
4:     else
5:         L[r, c] ← 0

6: kernel Merge(I, L, r, c)
7:     if I[r, c] = 1 then
8:         index ← LinearIndex(r, c)
9:         for all k_i ← Neighbours(index) do
10:            if k_i < index and I[k_i] = 1 then
11:                Union(L, index, k_i)

12: kernel Analysis(I, L, r, c)
13:    if I[r, c] = 1 then
14:        L[r, c] ← Find(L, LinearIndex(r, c)) + 1
```

During `Merge` phase, each thread working on a foreground pixel analyzes its neighbourhood, and for every foreground neighbour performs a `Union` with the active pixel. In the pseudo code, the *neighbours* function is used to get the neighbours of a pixel, which depend on the chosen connectivity.

After `Merge` phase the `Analysis` kernel performs the flattening of trees, completing the labeling task. The entire algorithm is first performed on rectangular blocks the image is divided into. Then, `Merge` kernel is performed on border pixels only, and a final `Analysis` is launched over the whole image. The original algorithm uses 4-connectivity, and we extended it to 8-connectivity by simply adding the diagonal directions to the neighbourhood of a pixel, in `Merge` kernel.

### Komura Equivalence Algorithm

The Komura Equivalence algorithm [35], which employs a 4-connectivity, introduces additional improvements to the UF algorithm.

**Algorithm 15** 4-connectivity Komura equivalence algorithm kernels. $I$ is input image and $L$ is labels matrix. Checks on image borders are not showed.

```
 1: kernel INITIALIZATION(I, L, r, c)
 2:     if I[r, c] = 0 then
 3:         L[r, c] ← 0
 4:     else
 5:         index ← LinearIndex(r, c)
 6:         label ← index
 7:         for all k_i ← Neighbours(index) do
 8:             if k_i < label and I[k_i] = 1 then
 9:                 label ← k_i
10:         L[r, c] ← label + 1

11: kernel REDUCTION(I, L, r, c)
12:     if I[r, c] = 1 and I[r, c − 1] = 1 then
13:         a ← LinearIndex(r, c)
14:         b ← LinearIndex(r, c − 1)
15:         Reduce(L, a, b)

16: procedure REDUCE(L, a, b)
17:     a ← Find(L, a)
18:     b ← Find(L, b)
19:     done ← (a = b)
20:     if a > b then
21:         Swap(a, b)
22:     while done = false do
23:         old ← atomicMin(&L[b], a + 1)
24:         if old = a + 1 then
25:             done ← true
26:         else if old > a + 1 then
27:             b ← old − 1
28:         else if old < a + 1 then
29:             b ← a
30:             a ← old − 1
```

This method consists of four steps: `Initialization`, `Analysis`, `Reduction`, and `Analysis` again, which are detailed in the following.

The `Initialization` kernel, shown at line 1 of Algorithm 15, sets the label of each pixel with the smallest linear address of its neighbours, avoiding the commonly used write operation which initializes the output image with increasing labels. Since the smallest address is chosen, north pixel is prioritized over west one. Same as Union-Find, in our implementation we add 1 to each label, to make sure that foreground pixels always are assigned positive values. Background pixels are given label 0 instead. This small change also slightly affects the other kernels.

The `Analysis` kernel is equal to the union-find procedure, and achieves the goal of collapsing the trees created in the previous step to their roots.

In `Reduction` kernel, shown at line 11 of Algorithm 15, every thread merges the active pixel tree with the one containing the pixel to the west, in the case it is foreground. Indeed, if both north and west pixels are fore-

ground, only north has been chosen during `Initialization` phase. The merging is performed by the *reduce* procedure, which has the same effect of *union*.

### 6.2.1 Proposed Algorithm

Our new algorithm is an adaptation of KE to a 8-connectivity scenario. It keeps the same structure as the original 4-connectivity variation—`Initialization`, `Analysis` and `Reduction` kernels are preserved. A few changes must be introduced to the operations these kernels perform though.

The `Initialization` kernel has the same structure as the 4-connectivity variation, but the *neighbours* function must also return diagonal neighbours. Since we still assign the smallest neighbour address, the priority order is north-west→north→north-east→west.

The `Analysis` kernel is the same as the 4-connectivity variation. Its job of following the equivalence chains to the root of trees is not indeed tied to pixel connectivity.

The `Reduction` kernel is still responsible for merging trees that result separate after the first two phases, but are connected together by at least a couple of foreground pixels nonetheless. This occurrence is rather common, considering that in the `Initialization` phase each pixel can only choose one among possibly four different neighbour indexes. Each possible neighbourhood configuration of a foreground pixel is shown in Fig. 6.2. Configurations in Fig. 6.2a do not need additional merging between pixel trees. In fact, connected pixels in the mask have already been linked together in `Initialization` phase. Conversely, crosses in Fig. 6.2b identify merging operations that must be performed between the tree containing the active pixel and the one containing the pixel to the west ($f$) or to the north-east ($g$ and $h$). Those two trees may have already been joined together by another pixel outside from the neighbourhood mask, but since we do not know it, we must join them anyway. A further exploration outside of the mask is not worth the effort, because of the large amount of memory accesses it would introduce. Each configuration in Fig. 6.2c exhibit one or more couple of trees that need to be merged as well. Nevertheless, each of those tree connections also show up in the neighbourhood mask of at least another foreground pixel, marked with a star (*e.g.* in $i$, the connection between the west pixel and the north-west pixel also appears in the neighbourhood mask of the west pixel). This means that, when we face a

mask configuration included in the aforementioned set, we are not forced to join neighbour trees. Indeed, other threads can perform the same operation. Consequently, to save unnecessary calls to the `Reduce` function, each thread in `Reduction` kernel only joins together connected trees if the neighbour configuration of the active pixel belongs to the *merge required* set, because those are the only cases we cannot delegate the merging operation to other threads. We use a small decision tree to identify said configurations while limiting the number of memory accesses to a minimum. Actual memory accesses range from a minimum of 1 to a maximum of 4 per thread, but their cost is compensated by the fact that `Reduce` function, which in turn performs a variable number of memory accesses, is only called on 3 neighbourhood configurations out of 16 possibilities.

We also introduced another slight improvement to `Reduction` kernel. As Playne noticed in [86], the original `Reduce` function always performs at least two *atomicMin*, even in the case no other thread interfered. We thus replaced the `Reduce` procedure with `Union`, which produces the same result without employing unnecessary atomic operations. The pseudo code of `Reduction` is showed in Algorithm 16. After the `Reduction` step, equally to the 4-connectivity variation, a final `Analysis` is needed to assign every pixel the root label of its tree.

**Algorithm 16** 8-connectivity Komura equivalence reduction kernel - $r$ and $c$ are thread row and column, $I$ is input image and $L$ is labels matrix. Checks on image borders are not showed.

```
1: kernel REDUCTION(I, L, r, c)
2:     if I[r, c] = 1 and I[r − 1, c] = 0 then
3:         k ← LinearIndex(r, c)
4:         NW ← I[r − 1, c − 1]
5:         if NW = 1 and I[r − 1, c + 1] = 1 then
6:             Union(L, k, LinearIndex(r − 1, c + 1))
7:         if NW = 0 and I[r, c − 1] = 1 then
8:             Union(L, k, LinearIndex(r, c − 1))
```

## 6.3 How Does Connected Components Labeling with Decision Trees Perform on GPUs?

In this Section the problem of Connected Components Labeling in binary images using Graphic Processing Units is tackled by a different perspective. In the last decade, many novel algorithms have been released, specifically designed for GPUs. Because CCL literature concerning sequential algorithms is very rich, and includes many efficient solutions, designers of parallel algorithms were often inspired by techniques that had already proved successful in a sequential environment, such as the union-find paradigm for solving equivalences between provisional labels. However, the use of decision trees to minimize memory accesses, which is one of the main feature of the best performing sequential algorithms (Chapter 3), was never taken into account when designing parallel CCL solutions. In fact, branches in the code tend to cause thread divergence, which usually leads to inefficiency. Anyway, this consideration does not necessarily apply to every possible scenario. Are we sure that the advantages of decision trees do not compensate for the cost of thread divergence? In order to answer this question, we chose three sequential CCL algorithms, which employ decision trees as the cornerstone of their strategy, and we built a data-parallel version of each of them. Experimental tests on real case datasets show that, in most cases, these solutions outperform state-of-the-art algorithms, thus demonstrating the effectiveness of decision trees also in a parallel environment.

### 6.3.1 Introduction

On GPUs, threads are grouped into packets (warps) and run on *single-instruction, multiple-data* (SIMD) units. This structure, called SIMT (*single-instruction, multiple threads*) by NVIDIA, allows to execute the same instruction on multiple threads in parallel, thus offering a potential efficiency advantage [134]. It is commonly known that most sequential programs, when ported on GPU, break the parallel execution model. Indeed, only applications characterized by regular control flow and memory access patterns can benefit from this architecture [135].

During execution, each processing element performs the same procedure

(kernel) on different data. All cores in a warp run like lock-step at same instruction, but next instruction can be fetched only when the previous one has been completed by all threads. If an instruction requires different amounts of time in different threads, such as when branches cause different execution flows, then all threads have to wait, decreasing the efficiency of the lock-step. This is the reason why intrinsically sequential algorithms must be redesigned to reduce/remove branches and fit GPU logic. But is this always necessary? Would this always improve performance? In this Chapter, focusing on the connected component labeling problem, we demonstrate that the best performing sequential algorithms can be easily implemented on GPU without changing their nature, and on real case scenarios they may perform significantly better than state-of-the-art algorithms specifically designed for GPUs.

### 6.3.2 Adapting Tree-Based Algorithms to GPUs

We adapt SAUF, BBDT and DRAG, described in Chapter 3 and Chapter 4, to a parallel environment, thus producing CUDA based CCL algorithms, that we call C-SAUF, C-BBDT and C-DRAG.

A GPU algorithm consists of a sequence of kernels, *i.e.*, procedures run by multiple threads of execution at the same time. In order to transform the aforementioned sequential algorithms into parallel ones, the three steps of which they are composed (*first scan*, *flattening*, and *second scan*) must be translated into appropriate kernels. In each of those steps, a certain operation is repeated over every element of a sequence. Thus, a naive parallel version consists in a concurrent execution of the same operation over the whole sequence. Un-

**Algorithm 17** Summary of algorithms kernels. $I$ and $L$ are input and output images. The pixel (or block) on which a thread works is denoted as $x$.

```
1:  kernel InitializationKernel(L)
2:      L[id_x] ← id_x

3:  kernel MergeKernel(I, L)
4:      DecisionTree(Mask(x))

5:  kernel CompressionKernel(L)
6:      L[id_x] ← Find(L, id_x)

7:  kernel FinalLabelingKernel(L)
8:      label ← L[id_x]
9:      for all a ∈ Block(x) do
10:         if I(a) = 1 then
11:             L(a) ← label
12:         else
13:             L(a) ← 0
```

fortunately, the *first scan* cannot be translated in such a simple way, because of its inherently sequential nature: when thread $t_x$ runs, working on pixel $x$, every foreground pixel in the neighborhood mask must already have a label. To address this issue, we assign an initial label to each foreground pixel, equal to its raster index ($id_x$). This choice has two important consequences. First, we can observe that there is no need to store provisional labels in the output image $L$, because calculating them is trivial. So, until *second scan*, $L$ can be used as the union-find structure, thus removing the need to allocate additional memory for $P$. In fact, in our parallel algorithms, $L \equiv P$. The second consequence is that the *first scan* loses the aim of assigning provisional labels, and it is only required to record equivalences. Its job is performed by two different kernels: `Initialization` and `Merge`.

The first one initializes the union-find array $L$. Of course, at the beginning, every label is the root of a distinct tree. Thus, in this kernel, thread $t_x$ performs $L[id_x] \leftarrow id_x$.

The second kernel, instead, deals with the recording of equivalences between labels. During execution, thread $t_x$ traverses a decision tree in order to decide which action needs to be performed, while minimizing the average amount of memory accesses. When no neighbors of the scanning mask are foreground, nothing needs to be done. In all other cases, the current label needs to be merged with those of connected pixels, with the *union* procedure. Moreover, the implementation of *union* proposed in Algorithm 1 requires to introduce atomic operations to deal with the concurrent execution.

Then, it is easy to parallelize the *flattening* step: it translates into a kernel (`Compression`) in which thread $t_x$ performs $L[id_x] \leftarrow$ `Find(L, id_x)` to link each provisional label to the representative of its union-find tree.

The last step of sequential algorithms is the *second scan*, which updates labels in the output image $L$. A large part of the job of *second scan* is not necessary in our parallel algorithms, because `Compression` kernel already solves label equivalences directly in the output image. In the case of C-SAUF, increasing foreground labels by one is the only remaining operation to perform, in order to ensure that connected components labels are positive numbers different from background. We avoid a specific kernel for this, shifting labels of foreground pixels by 1 since the beginning of the algorithm. This trick requires small changes to union-find functions. For

C-BBDT and C-DRAG, a final processing of $L$ is required to copy the label assigned to each block into its foreground pixels. This job is performed in `FinalLabeling` kernel. Table 6.2 sums up the structure of the proposed algorithms, while Algorithm 17 provides a possible implementation of the described kernels.

Table 6.2: Kernel composition of the proposed CUDA algorithms.

| Kernel | C-SAUF | C-BBDT | C-DRAG | Description |
|---|---|---|---|---|
| Initialization | ✓ | ✓ | ✓ | Creates starting *union-find* trees |
| Merge | ✓ | ✓ | ✓ | Merges trees of equivalent labels |
| Compression | ✓ | ✓ | ✓ | Flattens trees |
| FinalLabeling | | ✓ | ✓ | Copies block labels into pixels |

## 6.4 The Block-Based Approach on GPUs

In this Section, two new 8-connectivity CCL algorithms are proposed, namely Block-based Union-Find (BUF) and Block-based Komura Equivalence (BKE). These algorithms optimize existing GPU solutions introducing a block-based approach. Extensions for three-dimensional datasets are also discussed.

### 6.4.1 Introduction

Unfortunately, CCL is not as easy to parallelize as many other image processing tasks. Being it essentially a graph theory problem, algorithms need to perform graph traversal at a certain degree, which is an inherently sequential operation. For this reason, CPU and GPU algorithms usually have comparable performance [86]. However, the existence of efficient data parallel algorithms is valuable for applications that entirely run on GPU, allowing to remove the need for data transfers between host and device memory.

In this Section, we propose two new 8-connectivity GPU-based connected components labeling methods that improve previously proposed algorithms by taking advantage of the 2×2 block-based approach originally presented in [29] for sequential algorithms. This allows to drastically reduce the amount of memory accesses, thus improving overall performance.

In previous work [85], the 2×2 blocks were applied to the iterative GPU algorithm presented in [126] and improved in [127]. However, the chosen algorithm was rather slow compared to other approaches [33, 35]. Moreover, the benefit introduced in [127] was partially lessened by an increased allocation time, caused by the need for additional data structures to record information about blocks connectivity and blocks labels. We managed to adapt the block-based approach to the direct and more efficient algorithms proposed in [35] and [33]. Moreover, we removed the need for additional data structures, in order to minimize the amount of time spent for allocating and deallocating device memory. The results are two extremely fast solutions, able to improve state-of-the-art over both real case and synthetically generated datasets. Variations of our proposals are also described, meant to perform CCL on three-dimensional volumes.

### 6.4.2 Preliminaries

Our proposals are based on three different strategies, some of which have already been described in Section 6.1. Anyway, their fundamental characteristics are summarized here to simplify the reading.

#### The Union-Find algorithm

As said, UF is the first algorithm exploiting union-find on GPU. The original version of the algorithm employs 4-connectivity, but it can be easily extended to 8-connectivity [36]. The algorithm is based on three kernels: `Initialization`, `Merge` and `Compression`. An example of execution is depicted in Fig. 6.4. Each kernel runs on a number of threads equal to the image size, and each thread is assigned a pixel, which we will refer to as $x$. The union-find trees are coded in the output label image $L$.

During `Initialization`, every foreground pixel $p$ in the output image $L$ is initialized with its *id*, which corresponds to its raster index increased by 1. More specifically, thread $t_p$ performs $L[p] \leftarrow id_p$. From the union-find point of view, this procedure corresponds to the creation of a separate tree for every pixel. Background pixels are set to 0 instead.

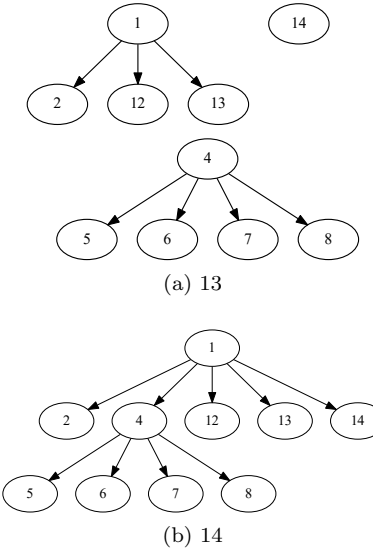



Figure 6.3: Change in the configuration of *union-find* trees operated by the thread working on pixel 14, and starting from the hypothetical provisional result of Fig. 6.4c. The thread checks its neighborhood mask in increasing raster index order. A *union* is performed between the single-node tree 14 and the tree rooted in 1. Then, a *union* between pixel 14 and pixel 4 causes the linking of the subtree starting at node 4 to the tree containing 14, rooted in 1.

During `Merge`, each thread working on a foreground pixel checks its neighbors, and for every foreground neighbor performs a *union* with $x$. Since *union* is a symmetric operation, there is no need to check the entire neighborhood. Therefore, only neighbors with a smaller raster index than $x$ are considered. These neighbors are identified by the mask depicted in Fig. 6.7a.

Fig. 6.3 shows the effects of `Merge` on the configuration of the union-find data structure, before and after the execution of thread 14, and starting from the provisional result of Fig. 6.4c. In this example, the thread operating on pixel 14 performs a *union* between the tree rooted in 14 and the trees rooted in 1 and 4.

After `Merge`, every connection between pixels in the image is reflected in the union-find structure, in the sense that every separate tree represents a connected component.

Finally, `Compression` kernel performs the compression of trees: every threads $t_p$ runs `Compress`$(L, p)$.

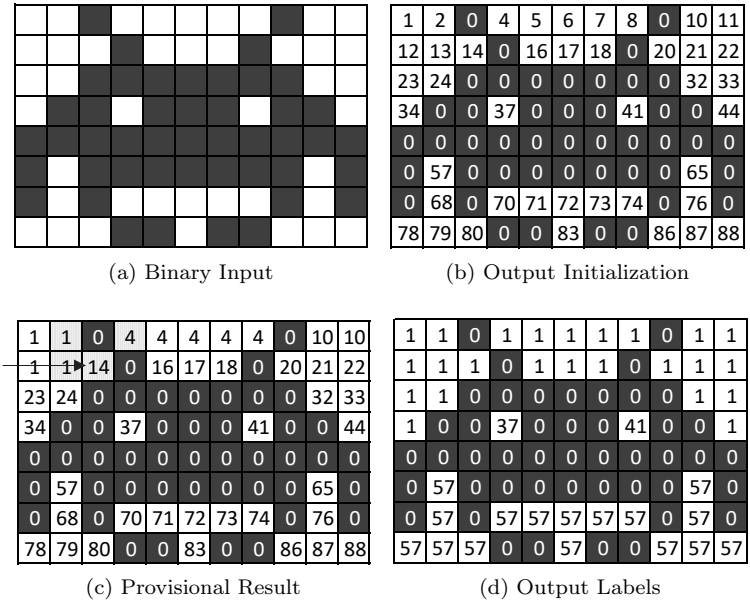
This operation makes sure that every pixel is assigned a label that corresponds to the raster index of the root of its tree. Thus, every pixel in the same CC ends up sharing the same label, and the labeling task is completed.

The aforementioned basic structure of the algorithm is enhanced with the addition of another common parallelization strategy, known as Tile Merging (TM). This means that the entire algorithm is first performed on rectangular blocks the image is divided into: this preliminary phase is called `LocalMerge`. Then, a `Merge` kernel is performed on border pixels only, and a final `Compression` is run over the whole image.

### 6.4.3 The Komura Equivalence Algorithm

The Komura Equivalence algorithm [35] is another GPU algorithm that can be seen as a variation of UF, which involves a more complex initialization in order to remove the need for one *union* per pixel later. It consists of four steps: `Initialization`, `Compression`, `Reduction`, and `Compression` again.

The main difference between KE an UF lies in the first kernel. Differently from UF, KE `Initialization` does not create single-node trees. Instead, every thread checks the neighborhood of $x$ in increasing raster index order, and the smallest foreground neighbor is set as the father node




(a) Binary Input (b) Output Initialization

(c) Provisional Result (d) Output Labels

Figure 6.4: Example of Union-Find execution on a negative Space Invaders character. The goal is to label differently the white areas. (b) is the expected labels image after `Initialization`. (c) is a provisional result of `Merge` kernel, under the exemplifying assumption that threads run in raster scan order, and the execution reached thread 14. The configuration of the union-find data structure before and after the execution of thread 14 is shown in Fig. 6.3. (d) is the labels image after the execution of the last kernel, `Compression`.

of $x$. Thus, this first phase aims at creating non single-node trees, that are flattened by the subsequent `Compression` kernel. UF and KE `Compression` kernels are exactly the same.

The `Reduction` kernel is a variation of `Merge`, that only performs a *union* between $x$ and foreground neighbors which were not chosen during `Initialization`. Analogously to UF, a final `Compression` is required for the flattening of the forest trees. Same as UF, the original KE is a 4-connectivity algorithm. It can as well be modified to deal with 8-

connectivity, adding the supplementary neighbors in both `Initialization` and `Reduction` kernels [36] (see Section 6.2 for more details).

### 6.4.4 The Block-Based Approach

Grana *et al.* noticed in [29] that, in the case of a two-dimensional image ($I_2$) and 8-connectivity, all foreground pixels within 2×2 blocks always share the same label (Fig. 6.5). This observation can be extended to volumes ($I_3$), where 26-connectivity implies that only one label can be assigned to foreground voxels of a 2×2×2 block.

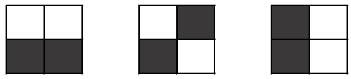


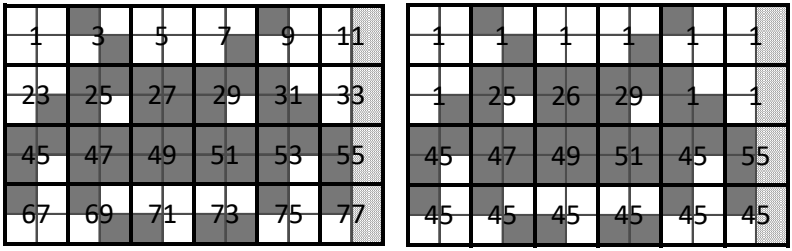
Figure 6.5: Examples of foreground pixels in a 2×2 block.

Consequently, when 8-connectivity (26-connectivity for volumes) is used, CCL can be applied to blocks, and block labels can be assigned to single pixels at the end of the algorithm. Such an approach usually has some advantages in terms of execution time, depending on the chosen algorithm. Considering UF, the use of blocks would divide the number of initial trees by 4 (8 for volumes), thus requiring considerably less *union* calls to achieve the final configuration. Moreover, the average depth of trees is reduced as well, speeding-up *find*. A similar reasoning can be applied to KE.

However, when blocks are considered in place of pixels, the definition of *connectivity* must change accordingly. We say that two blocks $P, Q$ are *connected* if one pixel of $P$ is connected to one pixel of $Q$:

$$P \diamond Q \Leftrightarrow \exists\, p \in P, q \in Q \mid\ p \diamond q \tag{6.1}$$

As a consequence, finding the connected neighbors of a block is more difficult than finding those of a single pixel, because the number of pixels to be checked is higher. Moreover, since the same internal pixel of a block can be responsible for connecting it to more than one neighbor blocks, a method that avoids multiple reads of the same pixel is valuable. Zavalishin *et al.*, who first applied blocks to GPU CCL, make use of pixel connectivity maps to find neighbor blocks while reading pixels only once [85].




(a) Provisional Labels (b) Final Labels

Figure 6.6: Example of Block-based Union-Find execution. (a) are the labels after `Initialization`. Every block has its own label, equal to the raster index of its top-left pixel. (b) are final block labels, after `Compression`. Blocks in the same connected component shares the same label, and the only remaining thing to do is to copy block labels into internal foreground pixels.

### 6.4.5 Proposed Algorithms

We propose two new 8-connectivity algorithms, which are optimized variations of UF and KE, obtained through the application of 2×2 blocks. We also extend our proposals to three-dimensional CCL.

#### Block-Based Union-Find

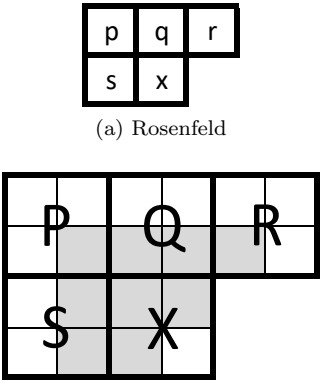
Block-based Union-Find (BUF) inherits the base structure of Union-Find (Section 6.4.2). The difference is that this algorithm works with block labels. In fact, every thread works on a 2×2 block, which we will refer to as the $X$ block. The algorithm implements the same kernels as UF, plus the additional `FinalLabeling`, which is needed to copy block labels into pixels. BUF kernels are depicted in Algorithms 18 and 19.

Until the end of the algorithm, block labels are stored in the output image, in the upper-left pixel of every block. In this way, we avoid the allocation of unnecessary device memory solely dedicated to block labels, which would be considerably time consuming.

In this case, `Merge` must be applied to blocks rather than to single pixels. The neighborhood mask used is depicted in Fig. 6.7.

Also in this case, it only contains blocks with a lower raster index than $X$. Since blocks connections are determined by those of their pixels, for every neighbor block of the mask we have to check whether some of its pixels are connected to some internal pixels of block $X$.

A naive approach that just checks each adjacent block one by one would require to read multiple times the internal pixels, but better alternatives exist. The method we adopted, based on the work by Zavalishin *et al.* [85], consists in a preliminary scan of pixels inside the block: for each foreground, its external neighbors are added to a set of pixels that must be checked in the subsequent step.



(a) Rosenfeld

(b) Block-based

Figure 6.7: Neighborhood mask used by the two-dimensional block-based algorithms described in this Section. Central block is $X$, and the connectivity between it and its neighbors depends on the value of grey pixels.

The aforementioned set is represented as a bitset $\mathcal{BS}$. Each pixel in a 4×4 square that encloses the $X$ block is given an index in the bitset, as reported in Fig. 6.8. Initially, every bit is set to 0.

When an internal pixel $p$ is read and recognized as foreground, external pixels $q$ such that $q \in \mathcal{N}_8(p)$ must have their corresponding bits in $\mathcal{BS}$ set to 1. To achieve this goal easily, the whole 3×3 square centered on $p$ is set accordingly by means of a bitmask (Fig. 6.8b).

Bitmask `0x777` is required to set neighbors of the top-left pixel inside block $X$. The other pixels bitmasks can be obtained in the following way: if the pixel is in the right column of the block, `0x777` is shifted one bit left. If the pixel is in the bottom row, the bitmask is shifted four bits left. The bottom-right pixel of $X$ is never responsible for connections between blocks inside the mask, so it is never used. To find out which neighbor blocks are connected to $X$, the `Merge` kernel must then check which pixels of $\mathcal{BS}$ are set and read their values. A *union* is performed between $X$ and connected blocks, same as what happens for single pixels in UF.

The BUF `Compression` kernel flattens the *union-find* trees by calling

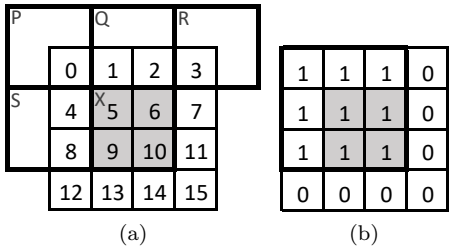



Figure 6.8: (a) shows how pixels in a 4×4 square centered on the $X$ block are numerated. These numbers correspond to the pixel position in the associated bitset $\mathcal{BS}$. Bits 0, 1, 2, 3, 4, and 8 are used to record whether the corresponding pixel is to be checked for determining blocks connectivity or not. The other bits are stored for convenience. (b) depicts the 3×3 bitmask (`0x777`) corresponding to the neighbors of the top-left internal pixel.

the `Compress` procedure defined in Section 2.2. We also propose a slight improvement of this kernel, that uses `InlineCompress` instead, thus producing the BUF_IC variation. The effects of `Merge` and `Compression` on an input image are depicted in Fig. 6.6.

`FinalLabeling` copies the label of each block into internal foreground pixels, thus producing the final output image. Background pixels are given label 0.

### Block-Based Komura Equivalence

The other new algorithm we propose is obtained from KE through the application of the same block-based approach. We therefore call it Block-based Komura Equivalence (BKE). BKE is quite similar to BUF in its structure. The main difference between the two is that BKE, same as KE, starts linking together connected blocks during `Initialization`. This means that the task of finding which blocks are connected to $X$ must be anticipated to the first kernel: the strategy to find connected blocks is the same described for BUF.

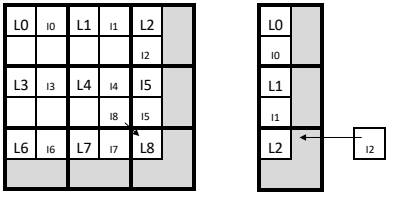
During `Initialization`, $X$ is linked to the connected neighbor block with the smallest raster index inside the mask, initializing the label of $X$ with the index of the connected neighbor.

The process of finding which blocks are connected to $X$ involves a high number of memory accesses. Since the information about connected blocks and foreground internal pixels is needed again during `Reduction` and `FinalLabeling`, we save it in a bitmapped byte, called *information byte*. Its structure is explained in Table 6.3.

Table 6.3: Meaning of the bitmapped byte used in Block-based Komura Equivalence to store information required by different kernels.

| *Bit* | *Meaning* |
|---|---|
| 0 | Top-left pixel is foreground |
| 1 | Top-right pixel is foreground |
| 2 | Bottom-left pixel is foreground |
| 3 | Bottom-right pixel is foreground |
| 4 | Not used |
| 5 | Block Q must undergo *union* |
| 6 | Block R must undergo *union* |
| 7 | Block S must undergo *union* |

In order to avoid the allocation of additional memory, the information byte is directly stored in the output image. The chosen location is the top-right pixel of every block, that would otherwise be unused until `FinalLabeling`, when the information byte is not necessary anymore. In the case that the image has an odd width, blocks in the last column do not have the top-right pixel, so we store connectivity information in the bottom-left pixel instead.




(a) 5x5 Image (b) 5x1 Image

Figure 6.9: Location of Label (L) and Information byte (I) of blocks in the output image, used by Block-based Komura Equivalence. In (b), the far-fetched case of a single column image with odd size is displayed. There, an additional byte is necessary to store I2.

If the image height is odd too, the last block of the last column is composed of the top-left pixel only. In that case, the connectivity information

**Algorithm 18** Block-based Union-Find `Initialization` and `Merge` kernels. $I$ and $L$ are input and output images, linearly stored in memory row-by-row. A padding can be added at the end of rows for alignment purpose, so *step* stores the effective length of rows in memory. Checks on image borders are not shown.

```
1:  kernel INITIALIZATION(L, step_L)
2:      r ← (threadIdx.y + blockIdx.y × blockDim.y) × 2
3:      c ← (threadIdx.x + blockIdx.x × blockDim.x) × 2
4:      x_L ← r × step_L + c
5:      L[x_L] ← x_L

6:  kernel MERGE(I, step_I, L, step_L)
7:      r ← (threadIdx.y + blockIdx.y × blockDim.y) × 2
8:      c ← (threadIdx.x + blockIdx.x × blockDim.x) × 2
9:      x_I ← r × step_I + c
10:     x_L ← r × step_L + c
11:     BS ← 0
12:     if I[x_I] = 1              then BS |=  0x777
13:     if I[x_I + 1] = 1          then BS |= (0x777 << 1)
14:     if I[x_I + step_I] = 1 then BS |= (0x777 << 4)
15:     if BS > 0 then
16:         if HasBit(BS, 0) and I[x_I − step_I − 1] then
17:             Union(L, x_L, x_L − 2 × step_L − 2)
18:         if  (HasBit(BS, 1) and I[x_I − step_I])  or
19:             (HasBit(BS, 2) and I[x_I − step_I + 1]) then
20:             Union(L, x_L, x_L − 2 × step_L)
21:         if HasBit(BS, 3) and I[x_I − step_I + 2] then
22:             Union(L, x_L, x_L − 2 × step_L + 2)
23:         if  (HasBit(BS, 4) and I[x_I − 1])  or
24:             (HasBit(BS, 8) and I[x_I + step_I − 1]) then
25:             Union(L, x_L, x_L − 2)
```

is stored in the bottom-right pixel of its neighbor $P$. If $P$ does not exists, *i.e.*, when the image is a single row or column with odd size, an extra byte is allocated.

So, we allocate additional memory only in degenerate cases, and com-

**Algorithm 19** Block-based Union-Find `Compression` and `FinalLabeling` kernels. $I$ and $L$ are input and output images, linearly stored in memory row-by-row. A padding can be added at the end of rows for alignment purpose, so *step* stores the effective length of rows in memory. Checks on image borders are not shown.

1: **kernel** COMPRESSION($L$, $step_L$)
2: $r \leftarrow (threadIdx.y + blockIdx.y \times blockDim.y) \times 2$
3: $c \leftarrow (threadIdx.x + blockIdx.x \times blockDim.x) \times 2$
4: $x_L \leftarrow r \times step_L + c$
5: `Compress`$(L, x_L)$

6: **kernel** FINALLABELING($I$, $step_I$, $L$, $step_L$)
7: $r \leftarrow (threadIdx.y + blockIdx.y \times blockDim.y) \times 2$
8: $c \leftarrow (threadIdx.x + blockIdx.x \times blockDim.x) \times 2$
9: $x_I \leftarrow r \times step_I + c$
10: $x_L \leftarrow r \times step_L + c$
11: $label \leftarrow L[x_L] + 1$
12: $L[x_L] \leftarrow label \times I[x_I]$
13: $L[x_L + 1] \leftarrow label \times I[x_I + 1]$
14: $L[x_L + step_L] \leftarrow label \times I[x_I + step_I]$
15: $L[x_L + step_L + 1] \leftarrow label \times I[x_I + step_I + 1]$

mon images never require extra data structure. Two possibilities of odd-sized images are exemplified in Fig. 6.9.

As for BUF, the `Compression` kernel flattens the union-find trees created in the previous phases to their roots. Two variations of this kernel exist, depending on the use of *InlineCompression*.

In the `Reduction` kernel the information byte is read for each block. Then, *union* operations are performed between connected neighbor blocks that were not linked to $X$ during `Initialization`. A subsequent `Compression`, followed by the `FinalLabeling`, completes the task. In `FinalLabeling`, the information byte is read again, in order to know whether each internal pixel should be assigned the block label or value 0, which corresponds to the background.

In both Block-based Union-Find and Komura Equivalence, the use of blocks drastically reduces the total amount of *union* and simplifies *find* operations, w.r.t. their pixel-based original versions. This optimization

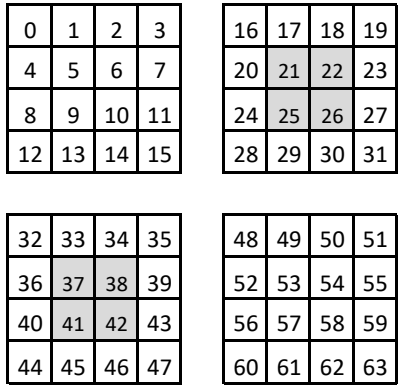



Figure 6.10: The figure shows how voxels in a 4×4×4 cube centered on the $X$ block are numerated. These numbers correspond to positions in the bitset, used for 3D. Only bits corresponding to internal voxels of blocks in the neighborhood mask are used.

allows to considerably reduce the number of memory accesses, which represents a bottleneck of parallel CCL algorithms.

### 3D Variations

BUF can be adapted to a 3D scenario without changes in its structure. The main difference is that voxels substitute pixels and blocks become 2×2×2 cubes.

The bitset used to represents neighbor voxels in the `Merge` kernel is larger than the 2D analogous. In fact, every voxel in a 4×4×4 cube must correspond to a bit in $\mathcal{BS}$. The 3D bitset is reported in Fig. 6.10. The remaining of the algorithm behaves similarly to its 2D counterpart, except for a slight optimization involving `Merge` and `FinalLabeling`.

Kernel `FinalLabeling` is responsible for copying each block label in the corresponding foreground internal voxels, and assigning label 0 to background ones. In the absence of information about internal voxels, this procedure requires 8 memory readings, one for each internal voxel of a 2×2×2 cube.

The information about internal voxels must be retrieved anyway in

`Merge`. So, we made `Merge` also responsible for writing which voxels are foreground in a bitmapped byte, and for storing it in the output image, similarly as what happens with the information byte of BKE. Then, in `FinalLabeling`, every thread just needs to read a single byte in order to know the internal configuration of $X$.

BKE is modified to suit 3D CCL in a similar way to BUF: 2×2×2 cubes are used in place of 2×2 squares, and the bitset that represents neighbor voxels is the same described above. The information byte used by the 2D variation becomes an information integer that requires 3 bytes in this case. In fact, it must be large enough to store both internal pixels values (8 bits) and neighbor blocks that need to undergo *union* with $X$ (13 bits). The overall behavior and the extra information storing strategy remain exactly the same as in the 2D counterpart.

## 6.5 Evaluation

According to the *Gestalt psychology* of perception, our senses operate the law of closure, perceiving objects as a whole even if they are loosely connected as for the 8-way connectivity [29]. This is the reason why many of the latest CCL proposals focus on 8-way connectivity. Based on this observation, and considering the amount of environments already covered by our analysis, we focus our experiments on the 8-connectivity for 2D images. On 3D volumes, 8-connectivity maps to 26-connectivity, that is employed in our analysis of 3D implementations.

As previously mentioned, many literature solutions considered in the comparison have no associate public source code, forcing us to re-implement them. All the experimental results have been obtained with the YACCLAB dataset and benchmarking framework, described in 3.4. All the algorithms are identified by their acronym already mentioned in Section 6.1.

### 6.5.1 Hardware and Software

The comparative evaluation is carried out on a single machine, running different operating systems and using different GPUs and compilers. Specifically, Windows 19.11.25508.2 and Ubuntu 16.04 LTS are considered, respectively combined with MSVC 19.11.25508.2 and GCC 9.3.0 compilers. In both cases, NVCC 11 has been used to compile CUDA files.

Table 6.4: Average run-time results in ms obtained under Ubuntu 16.04 LTS running a Quadro K2200 and a Quadro P1000 NVIDIA GPUs with GCC 9.3.0 and NVCC 11.0 compilers. The bold values represent the best algorithm on the specific environment/dataset.

| | | 2D Images | | | | | | | 3D Volumes | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *3DPeS* | *Fingerprints* | *Hamlet* | *Medical* | *MIRflickr* | *Tobacco800* | *XDOCS* | *Hilbert* | *Oasis* | *Mitochondria* |
| (a) *Quadro K2200* | M8DLS | 11.919 | 7.759 | 70.337 | 22.628 | 12.856 | 185.413 | 1766.380 | | | |
| | RASMUSSON | 5.231 | 4.060 | 117.980 | 8.970 | 5.122 | 255.681 | 3099.838 | | | |
| | 8DLS | 2.802 | 6.652 | 37.635 | 17.699 | 34.525 | 186.972 | 4787.276 | | | |
| | OLE | 0.619 | 0.449 | 5.052 | 2.654 | 0.549 | 7.597 | 35.112 | | | |
| | STAVA | 0.620 | 0.560 | 3.743 | 2.398 | 0.875 | 4.942 | 19.306 | | | |
| | LBUF | 0.392 | 0.252 | 2.911 | 1.716 | 0.261 | 3.858 | 15.749 | | | |
| | DLP | 0.446 | 0.204 | 3.046 | 1.561 | 0.298 | 4.803 | 17.248 | | | |
| | HA8 | 0.390 | 0.170 | 2.430 | 1.227 | 0.188 | 3.570 | 12.562 | | | |
| | KE | 0.348 | 0.198 | 2.608 | 1.467 | 0.226 | 3.607 | 14.501 | | | |
| | C_SAUF | 0.316 | 0.178 | 2.493 | 1.306 | 0.221 | 3.283 | 12.904 | | | |
| | C_BBDT | 0.312 | 0.181 | 2.140 | 1.045 | 0.216 | 2.903 | 10.930 | | | |
| | C_DRAG | 0.304 | 0.170 | 2.106 | 1.009 | 0.187 | 2.841 | 10.514 | | | |
| | BE | 0.678 | 0.429 | 3.307 | 1.650 | 0.557 | 4.305 | 16.492 | 3.721 | 7.998 | 49.271 |
| | UF | 0.385 | 0.253 | 3.012 | 1.967 | 0.363 | 4.105 | 16.811 | 3.877 | 23.407 | 151.858 |
| | BUF | 0.288 | 0.151 | 1.927 | 1.112 | 0.188 | 2.700 | 10.411 | **1.597** | **5.341** | **29.243** |
| | BKE | **0.279** | **0.139** | **1.801** | **0.951** | **0.144** | **2.623** | **9.189** | 2.269 | 7.142 | 80.461 |
| (b) *Quadro P1000* | M8DLS | 12.146 | 7.723 | 68.818 | 20.719 | 11.224 | 180.316 | 1722.653 | | | |
| | RASMUSSON | 3.077 | 2.424 | 65.854 | 4.973 | 3.027 | 141.074 | 1698.002 | | | |
| | 8DLS | 2.435 | 4.790 | 28.935 | 13.053 | 22.822 | 148.689 | 3899.968 | | | |
| | OLE | 0.668 | 0.343 | 4.137 | 2.036 | 0.425 | 6.342 | 30.048 | | | |
| | STAVA | 0.494 | 0.377 | 2.473 | 1.645 | 0.648 | 3.275 | 13.187 | | | |
| | LBUF | 0.329 | 0.162 | 2.000 | 1.233 | 0.175 | 2.618 | 11.427 | | | |
| | DLP | 0.357 | 0.101 | 1.815 | 0.944 | 0.166 | 2.907 | 10.811 | | | |
| | HA8 | 0.293 | 0.085 | 1.539 | 0.829 | 0.110 | 2.236 | 8.453 | | | |
| | KE | 0.322 | 0.117 | 1.916 | 1.100 | 0.150 | 2.704 | 11.525 | | | |
| | C_SAUF | 0.292 | 0.112 | 1.844 | 0.996 | 0.164 | 2.423 | 10.295 | | | |
| | C_BBDT | 0.280 | 0.092 | 1.541 | 0.773 | 0.129 | 2.126 | 8.493 | | | |
| | C_DRAG | 0.278 | 0.087 | 1.533 | 0.748 | 0.115 | 2.093 | 8.304 | | | |
| | BE | 0.440 | 0.278 | 2.431 | 1.314 | 0.305 | 3.432 | 13.424 | 2.852 | 6.529 | 39.324 |
| | UF | 0.296 | 0.140 | 1.927 | 1.277 | 0.224 | 2.628 | 11.249 | 2.925 | 20.943 | 125.191 |
| | BUF | 0.260 | 0.082 | 1.368 | 0.815 | 0.126 | 1.959 | 8.184 | **1.020** | **4.128** | **25.578** |
| | BKE | **0.251** | **0.066** | **1.266** | **0.686** | **0.081** | **1.911** | **7.219** | 1.427 | 5.244 | 50.989 |

The test machine is equipped with an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz (with $4 \times 32$ KB L1 cache, $4 \times 256$ KB L2 cache, 8 MB L3 cache) and two GPUs: a NVIDIA Quadro P1000 and a Quadro K2200, both with 4GB of GDDR5 GPU Memory and 640 CUDA Cores. The former has a Maxwell architecture, while the latter is based on Pascal architecture. NVIDIA Driver version 522.06 and 450.51 have been employed on Windows and Linux respectively.

In this tests, the kernel sensitivity to runtime parameters such as thread block and grid dimensions is not discussed. Although these parameters can significantly affect performance in CUDA programs, we stick on those provided in the original reference papers by verifying that such configurations provide the best results also on our environments.

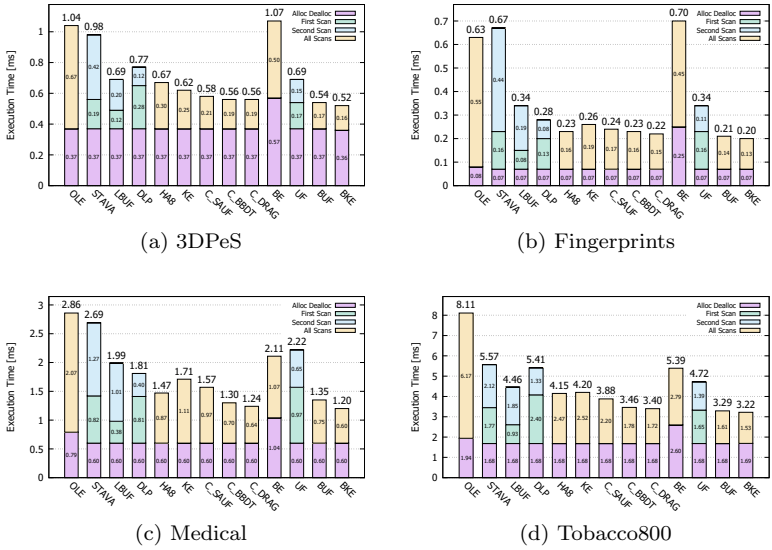
Table 6.5: Average run-time results in ms obtained under Windows 10 running a Quadro K2200 and a Quadro P1000 NVIDIA GPUs with MSVC 19.11.25508.2 and NVCC 11.8.89 compilers. The bold values represent the best algorithm on the specific environment/dataset.

| | | 2D Images | | | | | | | 3D Volumes | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | *3DPeS* | *Fingerprints* | *Hamlet* | *Medical* | *MIRflickr* | *Tobacco800* | *XDOCS* | *Hilbert* | *Oasis* | *Mitochondria* |
| (a) *Quadro K2200* | RASMUSSON | 5.537 | 4.428 | 117.100 | 9.169 | 5.463 | 249.293 | 2987.948 | | | |
| | 8DLS | 3.341 | 7.139 | 37.988 | 18.028 | 34.826 | 185.789 | 4732.362 | | | |
| | OLE | 0.958 | 0.612 | 5.215 | 2.806 | 0.698 | 7.985 | 34.102 | | | |
| | STAVA | 0.829 | 0.620 | 4.077 | 2.682 | 0.916 | 5.532 | 21.853 | | | |
| | LBUF | 0.602 | 0.306 | 3.214 | 1.978 | 0.293 | 4.423 | 18.105 | | | |
| | DLP | 0.662 | 0.260 | 3.325 | 1.807 | 0.328 | 5.375 | 19.542 | | | |
| | HA8 | 0.599 | 0.228 | 2.746 | 1.493 | 0.223 | 4.160 | 14.976 | | | |
| | KE | 0.565 | 0.254 | 2.929 | 1.735 | 0.260 | 4.207 | 16.954 | | | |
| | C_SAUF | 0.531 | 0.234 | 2.834 | 1.585 | 0.254 | 3.891 | 15.522 | | | |
| | C_BBDT | 0.514 | 0.224 | 2.424 | 1.318 | 0.240 | 3.461 | 13.087 | | | |
| | C_DRAG | 0.506 | 0.215 | 2.402 | 1.263 | 0.209 | 3.407 | 12.797 | | | |
| | BE | 1.020 | 0.694 | 3.781 | 2.029 | 0.924 | 5.283 | 18.765 | 4.786 | 9.540 | 90.104 |
| | UF | 0.598 | 0.307 | 3.312 | 2.213 | 0.391 | 4.684 | 19.115 | 3.492 | 17.798 | 132.286 |
| | BUF | 0.494 | 0.207 | 2.252 | 1.373 | 0.223 | 3.300 | 12.883 | **2.105** | **6.481** | **67.326** |
| | BKE | **0.487** | **0.196** | **2.127** | **1.216** | **0.178** | **3.214** | **11.688** | 2.714 | 8.259 | 102.297 |
| (b) *Quadro P1000* | M8DLS | 13.109 | 8.583 | 77.058 | 23.067 | 12.163 | 199.626 | 1898.248 | | | |
| | RASMUSSON | 3.569 | 2.888 | 67.363 | 5.387 | 3.467 | 140.871 | 1667.878 | | | |
| | 8DLS | 2.995 | 5.361 | 30.183 | 13.684 | 23.522 | 149.902 | 3888.757 | | | |
| | OLE | 0.991 | 0.482 | 4.390 | 2.244 | 0.568 | 6.718 | 28.536 | | | |
| | STAVA | 0.689 | 0.405 | 2.881 | 1.966 | 0.676 | 3.749 | 14.483 | | | |
| | LBUF | 0.515 | 0.179 | 2.386 | 1.532 | 0.192 | 3.077 | 12.718 | | | |
| | DLP | 0.544 | 0.120 | 2.183 | 1.228 | 0.177 | 3.344 | 12.023 | | | |
| | HA8 | 0.481 | 0.107 | 1.930 | 1.135 | 0.133 | 2.692 | 9.765 | | | |
| | KE | 0.511 | 0.137 | 2.320 | 1.408 | 0.169 | 3.182 | 12.886 | | | |
| | C_SAUF | 0.480 | 0.131 | 2.262 | 1.310 | 0.182 | 2.911 | 11.727 | | | |
| | C_BBDT | 0.468 | 0.106 | 1.927 | 1.091 | 0.145 | 2.591 | 9.748 | | | |
| | C_DRAG | 0.466 | 0.102 | 1.927 | 1.049 | 0.130 | 2.559 | 9.581 | | | |
| | BE | 0.775 | 0.465 | 2.952 | 1.703 | 0.481 | 4.151 | 14.324 | 3.459 | 7.023 | 58.013 |
| | UF | 0.483 | 0.156 | 2.311 | 1.568 | 0.237 | 3.083 | 12.498 | 2.379 | 13.939 | 95.871 |
| | BUF | 0.450 | 0.101 | 1.766 | 1.120 | 0.146 | 2.433 | 9.496 | **1.391** | **4.656** | **43.923** |
| | BKE | **0.442** | **0.089** | **1.672** | **0.997** | **0.104** | **2.394** | **8.595** | 1.753 | 5.605 | 65.939 |

### 6.5.2 Comparative Evaluation

Table 6.5 and 6.4 provide an overall comparison based on the total execution time targeting all the considered environments. Most of the algorithms lack a 3D implementation, so results on 3D volumes cannot be reported.

When focusing on 2D images, the overall performance of the algorithms and their mutual ranking are not affected by dataset intrinsic characteristics. Of course, the total execution time is deeply influenced by image size and complexity/number of connected components inside the image. XDOCS is certainly the hardest dataset to label and it holds the higher average execution time. Experimental results reveal that the advantages of the "M" variation on 8DLS is negligible on real-case datasets, unless images are very large and the majority of connected components is small,



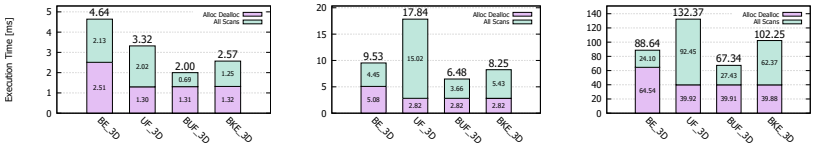

(a) 3DPeS (b) Fingerprints

(c) Medical (d) Tobacco800

Figure 6.11: Average run-time with steps test on 2D datasets under Windows using NVIDIA QUADRO K2200. Numbers are given in ms. Lower is better. Best viewed online.

as it happens within XDOCS. In all the other cases, the additional checks required by M8DLS ruin the overall algorithm performance. (M)8DLS and RASMUSSON algorithms are orders of magnitudes slower that all the others, so they are ignored in the rest of the analysis.

The remaining algorithms have more similar execution times; in order to have a deeper understanding of their behavior, Fig. 6.11 and Fig. 6.12 are reported for 2D and 3D datasets. In this figures the time needed for allocating data structures is separated from the one required by the global labeling procedure. Moreover, if an algorithm employs two clearly distinct phases to compute local labeling and perform tiles merging, the time required by each of them is displayed separately. The allocation time is the same for each strategy, except for BE and OLE that require additional data structures to the output image. OLE requires an additional byte to check whether the convergence has been reached or not. The




(a) Hilbert (b) Oasis (c) Mitochondria

Figure 6.12: Average run-time with steps test on 3D datasets. The experiment ran under Windows with NVIDIA QUADRO K2200. Lower is better. Best viewed online.

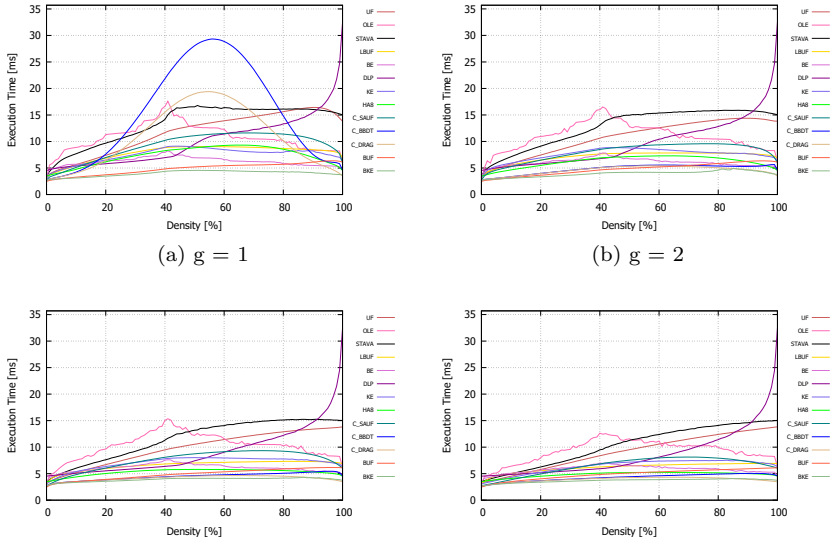
time required by `cudaMalloc` is not perfectly linear w.r.t. the amount of memory requested: in some cases, the additional allocation required for this single byte considerably increases the elapsed time. On the other hand, BE relies on additional matrices to store block adjacency and block labels, always requiring a data-dependent higher allocation time.

The experiments clearly demonstrate that allocating and freeing device memory represents a significant portion of the CCL total elapsed time, and therefore it should be avoided whenever possible, e.g., in embedded application or whenever the input image size is fixed.

The multiple iterations over the input image required by both OLE and STAVA make them the worst algorithms in most of the tested scenarios. The block scan approach proposed by BE reduces the operations (and thus the time) required by the global labeling w.r.t OLE at the expense of a longer allocation step. When the number of connected components is small, as it happens in 3DPeS and Fingerprints, the benefit of additional data structures are annihilated by the allocation time.

Direct algorithms, which perform a fixed number of kernel calls, tend to have better performance than iterative ones. UF is faster than OLE, STAVA and BE on almost all datasets (except for Medical), and the same goes for its optimization, LBUF and KE, the former using better block size and the latter with a more efficient initialization phase. DLP has a very similar approach to UF, since it also heavily relies on union-find, even if it implements it slightly differently; unsurprisingly, its performance is also comparable to UF.

The block-base approach proposed with BUF and BKE allows to lever-



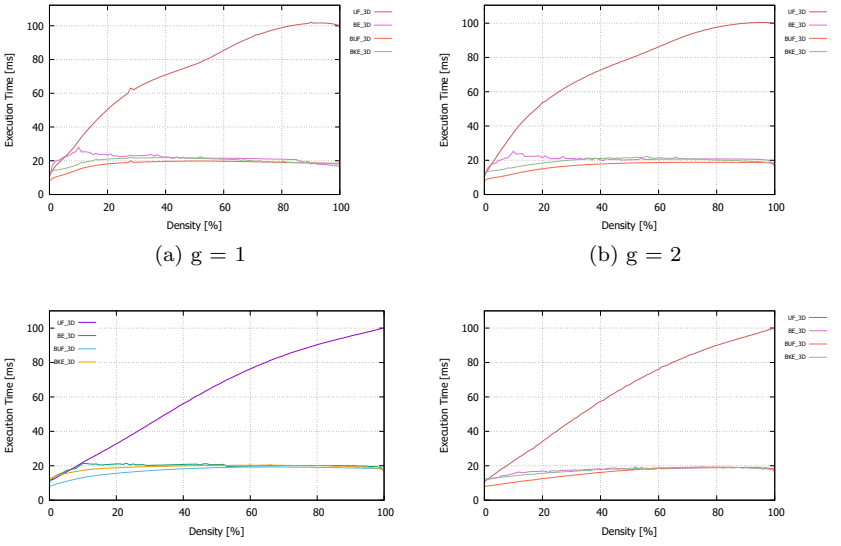

(a) g = 1 (b) g = 2

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

Figure 6.13: Granularity tests on 2D randomly generated images. Numbers are given in ms. Lower is better. Best viewed online.

age the advantages of blocks to furtherly improve UF and KE respectively without introducing allocation drawbacks, making the two algorithms the fastest available in literature for both 2D and 3D scenarios. BKE is available in OpenCV since version 4.6.0 for labeling connected components on GPU devices.

The algorithms based on decision trees, C_SAUF, C_BBDT, and C_DRAG, have performance close to the state of the art, with C_DRAG always being the best among the three. Although irregular patterns of execution slow down the thread scheduling and make the decision tree the worst thing to use in GPUs, when working with "natural" images the pattern within blocks are more or less uniform, making the thread divergence not really frequent. This is also confirmed by the granularity experiments reported in Fig. 6.13 and 6.14, on 2D and 3D randomly generated images. When granularity is 1, the computational time of both C_BBDT and C_DRAG




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

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

Figure 6.14: Granularity tests on 3D randomly generated images. Numbers are given in ms. Lower is better. Best viewed online.

explodes around 50% of density, in accordance with our expectation. These algorithms have performance close to state-of-the-art GPU-specific designs only when density is below 10% or over 90%. Indeed, at low/high densities the decision is taken by the first levels of the tree structures, saving a lot of memory accesses and without breaking the thread execution flow. Since images on which CCL is applied are usually in that range of density, this explains the previously observed behavior. When granularity grows, small portions of the image have irregular patterns requiring to explore deeper tree levels, while the vast majority tends to be all white or all black, again maximizing the tree performance.

Finally, thanks to an efficient combination of CUDA intrinsics to find runs of foreground pixels and union-find operations to merge runs of the same CC, HA8 performs close to state of the art on many datasets, especially when connected components tends to extend in the horizontal

direction (Fingerprints).

## 6.6 Concluding Remarks and Future Work

This Chapter started with a review of the state-of-the-art connected components labeling algorithms designed for GPUs and published in the last decade. Then, it introduced multiple novel strategies to speed up execution time, using diverse strategies: union-find, decision trees, code compression, block-based labels. Finally, all the different algorithms have been compared and their performance has been analyzed on different scenarios. Experimental results conducted on different operating systems, using different compilers and GPU architectures revealed that BKE is the best performing algorithm for 2D "real case" dataset including text images, fingerprints, medical images and surveillance dataset. When working on 3D volumes the performance of BKE are still impressive, but BUF demostrates better capabilities, especially when working with hard-to-label volumes such as the Hilbert fractal dataset. We also showed that algorithms specifically designed for sequential architecture can be extremely effective when adapted to GPUs.

The source code of all the analyzed algorithms is collected in the YACCLAB benchmark [78]. It is worth mentioning that we have been forced to re-implement many of the algorithm solution published in literature since the original implementation was not publicly available and some authors did not answer to our explicit requests. Anyway, the code is public so anyone can verify that our implementations are aligned with the description provided in the corresponding publications.

If we consider that the allocation time is "fixed" for a given environment and that the bare minimum of any CCL algorithm is the copy of the input into the output image (an operation that is not enough to complete the task, but strictly required) the reader can certainly spot that space for further optimization is really small, extremely so.

Future work should focus on the Connected Components Analysis (CCA) by smartly integrating the calculation of connected components statistics in the state-of-the art solutions.

Another open research direction is the optimization of CCL on embedded systems, including FPGA. Although specific algorithmic solution should be devised, reusing the core innovation of GPU-based solution could

benefit also such scenarios.

# Chapter 7

# Conclusion

The main aim of the research activities I carried out during my Ph.D. was the optimization of binary image processing algorithm, with a particular emphasis on connected components labeling.

In this thesis, all the successful techniques that improved CCL algorithms in the last decade have been combined in the novel Spaghetti Labeling algorithm, producing an extremely fast solution. The necessary steps required to achieve this result were an automatic generation of decision trees with state prediction, a solution to leverage all the savings obtainable at image edges, and finally a greedy algorithm allowing to convert decision forests into DAGs, thus reducing the machine code size more than a compiler could ever do. The proposed approach beats the results obtained by all compared algorithms in all settings.

Then, a novel framework was introduced, GRAPHGEN, that allows to automatically apply the most effective optimization strategies in literature to any problem modelled with decision tables. Starting from a simple definition of the problem the GRAPHGEN framework allows to generate optimal decision trees, to automatically apply state prediction, compression, and path-length optimization based on frequencies for any binary image processing problem. The output of the framework is the C++ source code implementing the optimized algorithm, the only requirement for the user is to model his needs as a set of rules. The effectiveness of GRAPHGEN-generated algorithms has been showcased on three different common binary image processing algorithms: connected components labeling, im-

age skeletonization and contour tracing. Furthermore, implementations of 3D algorithms have been demonstrated. When compared to state-of-the-art approaches, implementations using the GRAPHGEN-generated optimal DRAGs perform significantly better, on all the tasks.

Moreover, a generic heuristic approach for generating decision trees starting from an OR-decision table has been proposed, to overcome the limits given by the computational complexity of computing optimal decision trees. The resulting approach, Entropy Partitioning Decision Tree, EPDT in short, has been proven to generate near-optimal decision trees when applied to connected components labeling. Through the proposed heuristic it was possible to extend the block-based approach also to 3D volumes, significantly improving the state of the art on the field.

The focus then shifted toward the task of CCL on bitonal images. Three new algorithms have been presented: the first one has been generated using GRAPHGEN, and allows to label 4 pixels at once by expanding the block-based approach with the concept of macroblock. The other two proposals, on the other hand, are based on fast hardware-implemented Find First Set instructions to efficiently retrieve runs from a bitonal input. Experiments conducted over a collection of real world datasets demonstrate that these proposals outperform the fastest CCL algorithms in literature, thus representing the new state of the art for connecting components labeling on bitonal images.

Finally, the field of CCL on GPUs is deeply explored, starting with a comprehensive review of the works published in the last decade. Then, novel solutions have been proposed, inspired by the most successful techniques used for sequential algorithms. First, the Komura Equivalence (KE) algorithm, originally designed for 4-connectivity, has been adapted to 8-connectivity and improved with the addition of a simple decision tree. Then, three famous CCL algorithms based on decision trees have been adapted to GPUs, producing very fast solutions competing with state of the art. Finally, two GPU CCL algorithms, Union-Find (UF) and KE, have been enhanced by a block-based variation. The results, called Block-based Union-Find (BUF) and Block-based Komura Equivalence (BKE), are able to considerably reduce the number of memory accesses and consequently drastically reduce execution time.

At the moment of writing, the CCL algorithms Spaghetti and BKE, proposed in this thesis, represent the state of the art for CCL on CPU and GPU respectively, and are included in OpenCV, an open-source computer

vision and machine learning software library.

## Publications and Achievements

The effort presented in this thesis has resulted in publications in international conferences and journals.

Among all the others, the work on Spaghetti labeling has resulted in a journal paper on "IEEE Transactions on Image Processing" and two conference papers, while the general automatic optimization strategy of GRAPHGEN has been published on "IEEE Transactions on Pattern Analysis and Machine Intelligence". The effort on parallel algorithms has produced a journal paper on "IEEE Transactions on Parallel and Distributed Systems", as well, and other conference papers. A comprehensive survey on GPU-based connected components labeling is currently under review.

Appendix A reports the complete list of my publications. As the reader will notice, some of them did not fall under the line of this thesis and were therefore not discussed in the previous Chapters. Many of these papers have been done in collaboration with other Ph.D. students and researchers from the lab in which I carried out my research activity. I have to thank all of them for their cooperation and for the work we did together.

# Bibliography


[1] 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. 1, 2

[2] 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. 15575–15592, 2019. 1, 2, 16

[3] 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*, pp. 234–241, Springer, 2015. 1

[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)*, pp. 2079–2088, 2019. 1

[5] 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)*, pp. 1657–1666, 2017. 1

[6] 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)*, (Sophia Antipolis, France), pp. 162–167, IEEE, 2018. 1

[7] 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*, pp. 2641–2650, 2019. 1

[8] 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)*, pp. 284–293, 2019. 1

[9] 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)*, pp. 264–272, 2019. 1

[10] 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)*, pp. 4767–4776, 2019. 1

[11] 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*, pp. 4433–4442, 2017. 1

[12] 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)*, pp. 547–562, 2018. 1, 2

[13] 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)*, pp. 1019–1028, 2019. 1

[14] G. Máttyus, W. Luo, and R. Urtasun, "DeepRoadMapper: Extracting Road Topology from Aerial Images," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 3438–3446, 2017. 1, 2

[15] 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. 1

[16] 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. 1

[17] E. Ristani, F. Solera, R. Zou, R. Cucchiara, and C. Tomasi, "Performance measures and a data set for multi-target, multi-camera tracking," in *European Conference on Computer Vision*, pp. 17–35, Springer, 2016. 1

[18] 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)*, pp. 6064–6073, 2019. 1

[19] 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*, pp. 9416–9425, 2018. 1, 2

[20] 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. 1

[21] 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)*, pp. 899–908, 2019. 1

[22] 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. 1

[23] 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. 2

[24] J. Khodadoust and A. M. Khodadoust, "Fingerprint indexing based on minutiae pairs and convex core point," *Pattern Recognition*, vol. 67, pp. 110–126, 2017. 2, 66

[25] F. Uslu and A. A. Bharath, "A recursive Bayesian approach to describe retinal vasculature geometry," *Pattern Recognition*, vol. 87, pp. 157–169, 2019. 2, 66

[26] 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. 2, 66

[27] 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, pp. 1439–1458, Oct 2019. 2

[28] 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. 2

[29] 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. 9, 16, 18, 23, 45, 50, 56, 80, 85, 92, 93, 94, 97, 116, 117, 124, 142, 146, 154

[30] M. B. Dillencourt, H. Samet, and M. Tamminen, "A General Approach to Connected-Component Labeling for Arbitrary Image Representations," *Journal of the ACM*, vol. 39, no. 2, pp. 253–280, 1992. 10, 132

[31] 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. 10, 12, 16, 18, 84, 98, 103

[32] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013. 11

[33] V. M. Oliveira and R. A. Lotufo, "A Study on Connected Components Labeling algorithms using GPUs," in *SIBGRAPI*, vol. 3, p. 4, 2010. 11, 12, 116, 124, 131, 132, 142

[34] K. Yonehara and K. Aizawa, "A Line-Based Connected Component Labeling Algorithm Using GPUs," in *2015 Third International Symposium on Computing and Networking (CANDAR)*, pp. 341–345, IEEE, 2015. 11, 12, 126, 131

[35] 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. 11, 127, 131, 134, 142, 144

[36] 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)*, (Sophia Antipolis, France), pp. 175–180, IEEE, 2018. 11, 131, 143, 146

[37] 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. 11, 40, 115, 128, 131

[38] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Communications of the ACM*, vol. 7, pp. 301–303, May 1964. 12, 17

[39] R. E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm," *Journal of the ACM (JACM)*, vol. 22, pp. 215–225, Apr. 1975. 12

[40] L. He, Y. Chao, and K. Suzuki, "A Run-Based Two-Scan Labeling Algorithm," *IEEE Transactions on Image Processing*, vol. 17, no. 5, pp. 749–756, 2008. 13, 101, 102

[41] E. W. Dijkstra, *A discipline of programming*. Prentice-Hall Englewood Cliffs, N.J, 1976. 13, 45, 84

[42] M. Patwary, J. Blair, and F. Manne, "Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure," *Experimental Algorithms*, pp. 411–423, 2010. 13

[43] A. Rosenfeld and J. L. Pfaltz, "Sequential Operations in Digital Picture Processing," *Journal of the ACM*, vol. 13, pp. 471–494, Oct. 1966. 16, 17, 18

[44] A. Dubois and F. Charpillet, "Tracking Mobile Objects with Several Kinects using HMMs and Component Labelling," in *Workshop Assistance and Service Robotics in a human environment, International Conference on Intelligent Robots and Systems*, pp. 7–13, 2012. 16

[45] R. Cucchiara, C. Grana, A. Prati, and R. Vezzani, "Computer vision techniques for PDA accessibility of in-house video surveillance," in *First ACM SIGMM international workshop on Video surveillance*, pp. 87–97, ACM, 2003. 16

[46] A. Abramov, T. Kulvicius, F. Wörgötter, and B. Dellen, "Real-Time Image Segmentation on a GPU," in *Facing the multicore-challenge*, pp. 131–142, Springer, 2010. 16

[47] A. Körbes, G. B. Vitor, R. de Alencar Lotufo, and J. V. Ferreira, "Advances on Watershed Processing on GPU Architecture," in *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, pp. 260–271, Springer, 2011. 16

[48] A. Eklund, P. Dufort, M. Villani, and S. LaConte, "BROCCOLI: Software for fast fMRI analysis on many-core CPUs and GPUs," *Frontiers in neuroinformatics*, vol. 8, p. 24, 2014. 16

[49] H. V. Pham, B. Bhaduri, K. Tangella, C. Best-Popescu, and G. Popescu, "Real time blood testing using quantitative phase imaging," *PloS one*, vol. 8, no. 2, p. e55676, 2013. 16

[50] F. Pollastri, F. Bolelli, R. Paredes, and C. Grana, "Improving Skin Lesion Segmentation with Generative Adversarial Networks," in *2018 IEEE 31st International Symposium on Computer-Based Medical Systems (CBMS)*, pp. 442–443, IEEE, 2018. 16

[51] T. Lelore and F. Bouchara, "FAIR: A Fast Algorithm for Document Image Restoration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 8, pp. 2039–2048, 2013. 16

[52] F. Bolelli, "Indexing of Historical Document Images: Ad Hoc Dewarping Technique for Handwritten Text," in *Italian Research Conference on Digital Libraries (IRCDL)*, pp. 45–55, Springer, 2017. 16, 42

[53] T. Berka, "The Generalized Feed-forward Loop Motif: Definition, Detection and Statistical Significance," *Procedia Computer Science*, vol. 11, pp. 75–87, 2012. 16

[54] M. J. Dinneen, M. Khosravani, and A. Probert, "Using OpenCL for Implementing Simple Parallel Graph Algorithms," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, p. 1, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2011. 16

[55] S. Byna, M. F. Wehner, K. J. Wu, *et al.*, "Detecting atmospheric rivers in large climate datasets," in *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*, pp. 7–14, ACM, 2011. 16

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

[57] 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. 16, 18, 19, 20, 25, 40, 45, 50, 59, 80, 116

[58] S. Allegretti, F. Bolelli, M. Cancilla, F. Pollastri, L. Canalini, and C. Grana, "How does Connected Components Labeling with Decision Trees perform on GPUs?," in *18th International Conference on Computer Analysis of Images and Patterns*, pp. 39–51, Springer, 2019. 16, 131

[59] 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. 16

[60] F. Bolelli, L. Baraldi, M. Cancilla, and C. Grana, "Connected Components Labeling on DRAGs," in *2018 24th International Conference on Pattern Recognition (ICPR)*, pp. 121–126, 2018. 16, 20, 26, 32, 34, 45, 61, 84

[61] 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. 16, 19, 115

[62] 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. 16, 39, 98, 115, 131

[63] R. Haralick, "Some Neighborhood Operators," in *Real-Time Parallel Computing*, pp. 11–35, Springer, 1981. 16

[64] 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, 18, 45, 59

[65] A. Rosenfeld and A. Kak, *Digital picture processing.* No. v. 1 in Computer science and applied mathematics, Academic Press, 1982. 17

[66] 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. 17

[67] L. He, Y. Chao, and K. Suzuki, "A Linear-Time Two-Scan Labeling Algorithm," in *International Conference on Image Processing*, vol. 5, pp. 241–244, 2007. 18, 84

[68] X. Zhao, L. He, B. Yao, and Y. Chao, "A New Connected-Component Labeling Algorithm," *IEICE Transactions on Information and Systems*, vol. E98.D, no. 11, pp. 2013–2016, 2015. 18, 19, 20, 188, 190

[69] W.-Y. Chang and C.-C. Chiu, "An efficient scan algorithm for block-based connected component labeling," in *22nd Mediterranean Conference on Control and Automation*, pp. 1008–1013, 2014. 18, 40, 93

[70] D. Zhang, H. Ma, and L. Pan, "A Gamma-signal-regulated Connected Components Labeling Algorithm," *Pattern Recognition*, vol. 91, pp. 281–290, 2019. 18, 40, 115

[71] H. Samet and M. Tamminen, "An improved approach to connected component labeling of images," in *International Conference on Computer Vision And Pattern Recognition*, vol. 318, p. 312, 1986. 17

[72] 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. 19, 24, 26, 57, 58, 84, 97

[73] L. He, Y. Chao, and K. Suzuki, "An efficient first-scan method for label-equivalence-based labeling algorithms," *Pattern Recognition Letters*, vol. 31, no. 1, pp. 28–35, 2010. 19, 25

[74] C. Grana, L. Baraldi, and F. Bolelli, "Optimized Connected Components Labeling with Pixel Prediction," in *Advanced Concepts for Intelligent Vision Systems (ACIVS)*, pp. 431–440, Springer, 2016. 19, 25, 28, 45, 50, 59

[75] L. J. Schutte, "Survey of Decision Tables as a Problem Statement Technique," CSD-TR 80, 1973. 23

[76] H. Schumacher and K. C. Sevcik, "The Synthetic Approach to Decision Table Conversion," *Communications of the ACM*, vol. 19, pp. 343–351, June 1976. 24, 57

[77] S. R. Buss, "Alogtime Algorithms for Tree Isomorphism, Comparison, and Canonization," in *Kurt Gödel Colloquium on Computational Logic and Proof Theory*, pp. 18–33, Springer, 1997. 27, 62

[78] F. Bolelli, C. Grana, M. Cancilla, and S. Allegretti, "The YACCLAB Benchmark." `https://github.com/prittt/YACCLAB`. Accessed on 2023-01-10. 39, 52, 84, 98, 111, 116, 161

[79] 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)*, pp. 3109–3114, IEEE, 2016. 39, 63, 84, 98, 109

[80] 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. 229–244, 2018. 39, 45, 48, 50, 63, 84, 98

[81] J. Chen, K. Nonaka, H. Sankoh, R. Watanabe, H. Sabirin, and S. Naito, "Efficient Parallel Connected Component Labeling with a Coarse-to-fine Strategy," *IEEE Access*, vol. 6, pp. 55731–55740, 2018. 40

[82] T. Chabardès, P. Dokládal, and M. Bilodeau, "A labeling algorithm based on a forest of decision trees," *Journal of Real-Time Image Processing*, pp. 1527–1545, 2019. 40

[83] F. Diaz-del Rio, P. Sanchez-Cuevas, H. Molina-Abril, and P. Real, "Parallel Connected-Component-Labeling based on Homotopy Trees," *Pattern Recognition Letters*, 2019. 40

[84] A. Torralba and A. A. Efros, "Unbiased Look at Dataset Bias," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1521–1528, IEEE, 2011. 40

[85] 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. 40, 116, 124, 131, 142, 146, 148

[86] D. P. Playne and K. Hawick, "A New Algorithm for Parallel Connected-Component Labelling on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 1217–1230, June 2018. 40, 42, 43, 115, 137, 142

[87] M. J. Huiskes and M. S. Lew, "The MIR Flickr Retrieval Evaluation," in *International Conference on Multimedia Information Retrieval*, (New York, NY, USA), pp. 39–43, ACM, 2008. 41

[88] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979. 41, 42, 43

[89] 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. 41

[90] "The Hamlet Dataset." `http://www.gutenberg.org`. Accessed on 2023-01-10. 41

[91] G. Agam, S. Argamon, O. Frieder, D. Grossman, and D. Lewis, "The Complex Document Image Processing (CDIP) Test Collection Project." Illinois Institute of Technology, 2006. 41

[92] 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*, pp. 665–666, ACM, 2006. 41

[93] "The Legacy Tobacco Document Library (LTDL)." University of California, San Francisco, 2007. 41

[94] F. Bolelli, G. Borghi, and C. Grana, "Historical Handwritten Text Images Word Spotting Through Sliding Window Hog Features," in *19th International Conference on Image Analysis and Processing (ICIAP)*, pp. 729–738, Springer, 2017. 42

[95] F. Bolelli, G. Borghi, and C. Grana, "XDOCS: An Application to Index Historical Documents," in *Italian Research Conference on Digital Libraries (IRCDL)*, pp. 151–162, Springer, 2018. 42

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

[97] J. Sauvola and M. Pietikäinen, "Adaptive document image binarization," *Pattern recognition*, vol. 33, no. 2, pp. 225–236, 2000. 42

[98] 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*, pp. 59–64, ACM, 2011. 42

[99] 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. 43

[100] 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*, pp. 1987–1994, 2013. 43

[101] M. Matsumoto and T. Nishimura, "Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998. 44

[102] 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. 55, 97, 116

[103] F. Bolelli, C. Grana, and M. Soëcthing, "GRAPHGEN source-code and documentation." `https://github.com/prittt/graphgen`. Accessed on 2023-01-10. 55

[104] 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. 59

[105] E. S. Deutsch, "Thinning Algorithms on Rectangular, Hexagonal, and Triangular Arrays," *Communications of the ACM*, vol. 15, no. 9, pp. 827–837, 1972. 65

[106] L. Lam, S.-W. Lee, and C. Y. Suen, "Thinning Methodologies—A Comprehensive Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 14, no. 9, pp. 869–885, 1992. 65

[107] T. Y. 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. 66

[108] 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. 66, 68

[109] Z. Guo and R. W. Hall, "Parallel Thinning with Two-Subiteration Algorithms," *Communications of the ACM*, vol. 32, no. 3, pp. 359–373, 1989. 66

[110] "Source code of the THeBE benchmarking system." https://github.com/prittt/THeBE. Accessed on 2023-01-10. 69

[111] F. Bolelli and C. Grana, "Improving the Performance of Thinning Algorithms with Directed Rooted Acyclic Graphs," in *ICIAP*, vol. 11752, pp. 148–158, 2019. 69

[112] H. Freeman, "On the Encoding of Arbitrary Geometric Configurations," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 260–268, June 1961. 71

[113] 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. 71, 76

[114] F. Bolelli, S. Allegretti, and C. Grana, "BACCA source-code." https://github.com/prittt/bacca. Accessed on 2023-01-10. 76

[115] 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. 76

[116] 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. 78, 84

[117] L. Rokach and O. Maimon, "Top-Down Induction of Decision Trees Classifiers - A Survey," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 35, no. 4, pp. 476–487, 2005. 79

[118] J. R. Quinlan, "Induction of Decision Trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986. 79

[119] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Elsevier, 2014. 79

[120] W.-Y. Chang, C.-C. Chiu, and J.-H. Yang, "Block-Based Connected-Component Labeling Algorithm Using Binary Decision Trees," *Sensors*, vol. 15, no. 9, pp. 23763–23787, 2015. 93

[121] F. Bolelli, S. Allegretti, and C. Grana, "One DAG to Rule Them All," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–12, 2021. 115

[122] L. Cabaret, L. Lacassagne, and D. Etiemble, "Parallel light speed labeling: An efficient connected component labeling algorithm for multi-core processors," *Journal of Real-Time Image Processing*, vol. 15, no. 1, pp. 173–196, 2018. 115

[123] F. Lemaitre, A. Hennequin, and L. Lacassagne, "How to speed Connected Component Labeling up with SIMD RLE algorithms," in *Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing*, pp. 1–8, 2020. 115

[124] H. Samet, "Connected Component Labeling Using Quadtrees," *Journal of the ACM (JACM)*, vol. 28, no. 3, pp. 487–501, 1981. 116

[125] D. J. C. Santiago, T. I. Ren, G. D. Cavalcanti, and T. I. Jyh, "Efficient 2×2 block-based connected components labeling algorithms," in *2015 IEEE International Conference on Image Processing (ICIP)*, pp. 4818–4822, 2015. 116

[126] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel graph component labelling with GPUs and CUDA," *Parallel Computing*, vol. 36, no. 12, pp. 655–678, 2010. 118, 119, 131, 142

[127] 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. 120, 131, 142

[128] O. Šťava and B. Beneš, "Chapter 35 - Connected Component Labeling in CUDA," in *GPU Computing Gems Emerald Edition*

(W. mei W. Hwu, ed.), Applications of GPU Computing Series, pp. 569 – 581, Boston: Morgan Kaufmann, 2011. 120, 131

[129] P. Chen, H. Zhao, C. Tao, and H. Sang, "Block-run-based connected component labelling algorithm for gpgpu using shared memory," *Electronics Letters*, vol. 47, pp. 1309–1311(2), November 2011. 121, 131

[130] Y. Soh, H. Raja, Y. Hae, and I. Kim, "Fast Parallel Connected Component Labeling Algorithms Using CUDA Based on 8-directional Label Selection," *International Journal of Latest Research in Science and Technology*, vol. 3, pp. 187–190, 03 2014. 122, 131

[131] A. Rasmusson, T. Sørensen, and G. Ziegler, "Connected Components Labeling on the GPU with Generalization to Voronoi Diagrams and Signed Distance Fields," in *Advances in Visual Computing*, vol. 8033, pp. 206–215, 07 2013. 123, 131

[132] F. N. Paravecino and D. Kaeli, "Accelerated Connected Component Labeling Using CUDA Framework," in *Computer Vision and Graphics*, vol. 8671, 09 2014. 127, 131

[133] A. Hennequin, L. Lacassagne, L. Cabaret, and Q. Meunier, "A new Direct Connected Component Labeling and Analysis Algorithms for GPUs," in *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 76–81, IEEE, 2018. 128, 131

[134] J. Nickolls and W. J. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, no. 2, 2010. 138

[135] N. Brunie, S. Collange, and G. Diamos, "Simultaneous Branch and Warp Interweaving for Sustained GPU Performance," in *39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 49–60, June 2012. 138

# Appendix A

# List of Publications

This Section contains the list of research papers published during the Ph.D. period, as well as a pre-print which is currently under review. Some of them did not make it in the final thesis, either because they have been improved or replaced by a successive work, either because their topic did not overlap with the main flow of this thesis.
Content and experimental results published in some of these papers has been included in this thesis, with explicit permission given by the other authors.


[1] Stefano Allegretti, Federico Bolelli, Michele Cancilla, and Costantino Grana. Optimizing GPU-Based Connected Components Labeling Algorithms. In *2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS)*, pages 175–180. IEEE, December 2018.

[2] Stefano Allegretti, Federico Bolelli, Michele Cancilla, and Costantino Grana. A Block-Based Union-Find Algorithm to Label Connected Components on GPUs. In *Image Analysis and Processing - ICIAP 2019*, pages 271–281. Springer, September 2019.

[3] Stefano Allegretti, Federico Bolelli, Michele Cancilla, Federico Pollastri, Laura Canalini, and Costantino Grana. How does Connected Components Labeling with Decision Trees perform on GPUs? In *Computer Analysis of Images and Patterns*, pages 39–51. Springer, September 2019.

[4] Laura Canalini, Federico Pollastri, Federico Bolelli, Michele Cancilla, Stefano Allegretti, and Costantino Grana. Skin Lesion Segmentation Ensemble with Diverse Training Strategies. In *Computer Analysis of Images and Patterns*, pages 89–101. Springer, September 2019.

[5] Stefano Allegretti, Federico Bolelli, and Costantino Grana. Optimized Block-Based Algorithms to Label Connected Components on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, pages 423–438, 2020.

[6] Stefano Allegretti, Federico Bolelli, and Costantino Grana. A warp speed chain-code algorithm based on binary decision trees. In *2020 Joint 9th International Conference on Informatics, Electronics & Vision (ICIEV) and 2020 4th International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pages 1–8, 2020.

[7] Federico Bolelli, Stefano Allegretti, Lorenzo Baraldi, and Costantino Grana. Spaghetti Labeling: Directed Acyclic Graphs for Block-Based Connected Components Labeling. *IEEE Transactions on Image Processing*, pages 1999–2012, 2020.

[8] Stefano Allegretti, Federico Bolelli, Federico Pollastri, Sabrina Longhitano, Giovanni Pellacani, and Costantino Grana. Supporting Skin Lesion Diagnosis with Content-Based Image Retrieval. In *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, Jan 2021.

[9] Federico Bolelli, Stefano Allegretti, and Costantino Grana. A heuristic-based decision tree for connected components labeling of 3d volumes: Implementation and reproducibility notes. In *International Workshop on Reproducible Research in Pattern Recognition*, pages 139–145. Springer, 2021.

[10] Federico Bolelli, Stefano Allegretti, and Costantino Grana. One DAG to Rule Them All. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 1–12, 2021. 115

[11] Michele Cancilla, Laura Canalini, Federico Bolelli, Stefano Allegretti, Salvador Carrión, Roberto Paredes, Jon A Gómez, Simone Leo, Marco Enrico Piras, Luca Pireddu, et al. The deephealth toolkit:

a unified framework to boost biomedical applications. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 9881–9888. IEEE, 2021.

[12] Wonsang Lee, Stefano Allegretti, Federico Bolelli, and Costantino Grana. Fast Run-Based Connected Components Labeling for Bitonal Images. In *2021 Joint 10th International Conference on Informatics, Electronics and Vision (ICIEV) and 2021 5th International Conference on Imaging, Vision and Pattern Recognition (icIVPR)*, pages 1–8. IEEE, Aug 2021.

[13] Maximilian Söchting, Stefano Allegretti, Federico Bolelli, and Costantino Grana. A heuristic-based decision tree for connected components labeling of 3d volumes. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 7751–7758. IEEE, 2021.

[14] Federico Bolelli, Stefano Allegretti, and Costantino Grana. Connected Components Labeling on Bitonal Images. In *Image Analysis and Processing - ICIAP 2021*, pages 347–357. Springer, May 2022.

[15] Federico Bolelli, Stefano Allegretti, and Costantino Grana. Quest for speed: The epic saga of record-breaking on opencv connected components extraction. In Pier Luigi Mazzeo, Emanuele Frontoni, Stan Sclaroff, and Cosimo Distante, editors, *Image Analysis and Processing. ICIAP 2022 Workshops*, pages 107–118, Cham, 2022. Springer International Publishing.

[16] Marco Cipriano, Stefano Allegretti, Federico Bolelli, Mattia Di Bartolomeo, Federico Pollastri, Arrigo Pellacani, Paolo Minafra, Alexandre Anesi, and Costantino Grana. Deep Segmentation of the Mandibular Canal: a New 3D Annotated Dataset of CBCT Volumes. *IEEE Access*, pages 1–11, 2022.

[17] Marco Cipriano, Stefano Allegretti, Federico Bolelli, Federico Pollastri, and Costantino Grana. Improving Segmentation of the Inferior Alveolar Nerve through Deep Label Propagation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–10. IEEE, 2022.

[18] Stefano Allegretti, Federico Bolelli, and Costantino Grana. Connected Components Labeling on GPUs: a State of the Art Review with Code.

*IEEE Transactions on Parallel and Distributed Systems*, pages 1–18, 2023. *Under Review.*

# Appendix B

# Additional Experimental Results

Some of the experimental results discussed in the previous Chapters have been placed in this Appendix to make the reading flow of the thesis easier. They are here reported divided by Chapters.

**Spaghetti: Directed Acyclic Graphs for Block-Based Connected Components Labeling, 3**

Additional experimental results that allows to compare the Spaghetti algorithm with state-of-the-art algorithms from a different and exhaustive point of view are here reported. These tests have been performed on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz (6×32 KB L1 data cache, 6×32 KB L1 instruction cache, 6×256 KB L2 cache, and 12 MB of L3 cache) running Windows 10.0.17134 (64 bit) OS with both MSVC 19.15.26730 (Table B.1) and GCC 5.1.0 (Table B.2) compilers and Linux 4.18.0 (64 bit) OS with GCC 5.5.0 compiler (Table B.3).

This comparison highlights how the performance of an algorithm coupled with a label solver may significantly change with the environment. The RemSP label solver is not always the best choice for Spaghetti. On this specific architecture, the algorithm performs better with UF solver on Windows (both with MSVC and GCC) and with RemSP solver under Linux

Table B.1: Average run-time tests in ms on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz on Windows 10.0.17134 (64 bit) OS with MSVC 19.15.26730 compiler. Lower is better.

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

(with GCC). Additionally, in Table B.4 the same strategy of Spaghetti is also applied to the mask presented in [68],

which is identified by CTBE (CTB Enhanced to work with three lines). Applying the proposed strategy to this mask lowers the performance, because it is unable to consider the fact that all pixels in a single block share the same label, running the mask twice for blocks which are instead a single step for the block based mask. Even if our implementation is not the original one, it is possible to see that effectively CTBE improves CTB a little, as reported in [68].

Table B.2: Average run-time tests in ms on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz on Windows 10.0.17134 (64 bit) OS with GCC 5.1.0 compiler. Lower is better.

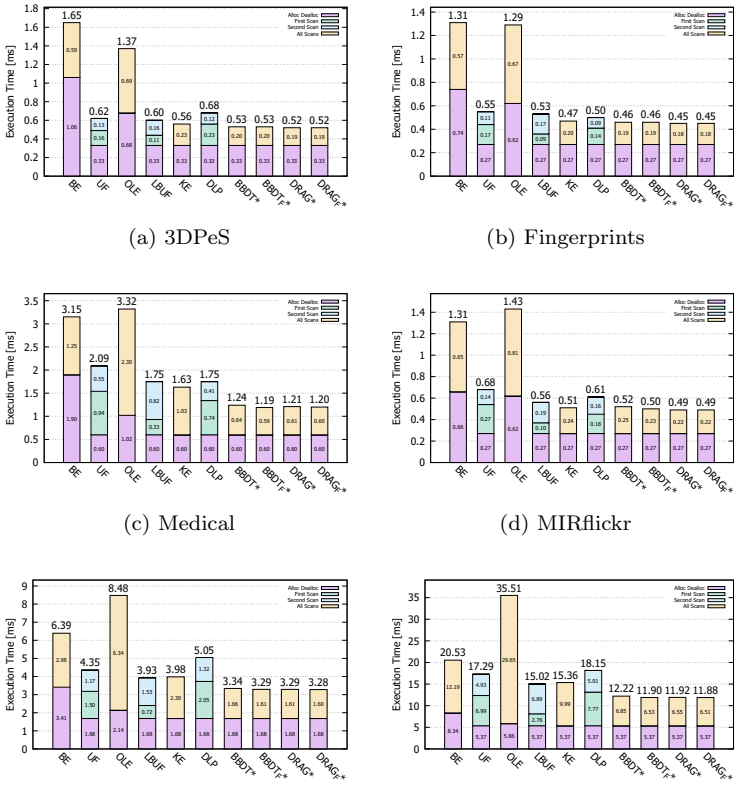
| | *3DPeS* | *Fingerprints* | *Medical* | *MIRflickr* | *Tobacco800* | *XDOCS* |
|---|---|---|---|---|---|---|
| SAUF_RemSP | 0.986 | 0.364 | 3.052 | 0.531 | 11.602 | 43.068 |
| SAUF_UF | 0.988 | 0.369 | 3.062 | 0.537 | 11.625 | 43.214 |
| SAUF_UFPC | 0.999 | 0.402 | 3.119 | 0.543 | 11.758 | 44.069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

Table B.3: Average run-time tests in ms on an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz on Linux 4.18.0 (64 bit) OS with GCC 5.5.0 compiler. Lower is better.

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

Table B.4: Average run-time tests for our algorithm applied to the mask described in [68]. Settings of Table B.1.

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




(a) 3DPeS

(b) Fingerprints

(c) Medical

(d) MIRflickr

(e) Tobacco800

(f) XDOCS

Figure B.1: Average run-time experimental results on 2D CCL algorithms on GPU in milliseconds. Results 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. The star identifies novel variations on previous algorithms generated thanks to GRAPHGEN. Lower is better.