University of Modena and Reggio Emilia
XXXII cycle of the international doctorate school in
Information and Communication Technologies

Doctor of Philosophy dissertation in
Computer Engineering and Science

Optimization of Binary Image Processing Algorithms

"One DAG to Rule Them All"

Federico Bolelli

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

Modena, 2020

Abstract

The procedure of making an algorithm more efficient in terms of memory requirements or execution time is called optimization and represents a crucial step in image and video processing. Usually, it is achieved with a trade-off between time and memory. Anyway, in many scenarios the total execution time required to complete a task is the most restrictive constraint. Binary image processing algorithms, for example, represent a fundamental pre- and post-processing operation in most of the state-of-the-art image and video analysis pipelines, even when they are based on deep learning techniques. For this reason, having a fast implementation is crucial, especially when these pipelines must be employed in real time scenarios in which compromising the output quality result or resort to more powerful hardware is not a choice. This thesis introduces and explores different approaches for the optimization of all the binary image processing algorithms that can be modeled with decision tables. There is a large amount of algorithms that can be defined in such a way: Connected Component Labeling, 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. Focusing on Connected Component Labeling, this thesis analyzes the state-of-the-art approaches for both sequential CPU-based and parallel CPU- and GPU-based environments, focusing on how to fairly measure performance. We then introduce novel approaches to further optimize such a kind of algorithms, showing how these optimization techniques can be generalized to boost the performance of any algorithm modeled with decision tables. A framework that allows to automatically apply the optimization strategies to a given problem is then presented. The framework, called GRAPHGEN,

takes a definition of the problem in terms of conditions to check and actions to be performed as input and it is able to produce the *C++* code including all the required optimizations as output.

When compared to existing approaches, the algorithms generated with GRAPHGEN perform significantly better than previous state-of-the-art algorithms, on real-world and synthetic datasets.

Review committee composed of:
Prof. Giuseppe Serra, Università degli Studi di Udine
Prof. Lamberto Ballan, Università degli Studi di Padova

To Elena

Contents

Contents **ix**

1 Introduction **1**

2 Literature Survey **9**
2.1 Connected Components Labeling 9
2.1.1 Sequential Algorithms 10
2.1.2 Parallel CPU- and GPU-based Algorithms 12
2.2 Image Skeletonization 14
2.3 Chain-Code 16

3 Mathematical Preliminaries **19**
3.1 Basic Notations and Definitions 19
3.2 The *Union-Find* Data Structure 21

4 Toward Reliable Experiments on Algorithms Performance **27**
4.1 Introduction 28
4.2 The Dataset 30
4.3 The Project 33
4.3.1 NULL labeling 39
4.3.2 *Union-Find* Templating 40
4.4 Available Algorithms 40
4.5 Experimental Results 43
4.6 Support for GPU and 3D CCL Algorithms 51
4.7 Conclusion 54

5 A New Paradigm for Sequential CCL Algorithms 55
5.1 Optimized State Prediction 55
5.1.1 Background 56
5.1.2 Method 57
5.1.3 Experimental Evaluation 61
5.1.4 Conclusion 63
5.2 Connected Components Labeling on DRAGs 64
5.2.1 Modelling CCL with Decision Trees 65
5.2.2 From Decision Trees to DRAGs 66
5.2.3 Experimental Results 70
5.2.4 Conclusion 70
5.3 Spaghetti: Directed Acyclic Graphs for Block-Based Connected Components Labeling 72
5.3.1 Background 72
5.3.2 Preliminaries 74
5.3.3 Outline of the Spaghetti Algorithm 78
5.3.4 Comparative Evaluation 89
5.3.5 Conclusion 96

6 CCL in Parallel Environments 97
6.1 Multi-Cores Architectures 97
6.1.1 Experimental Evaluation 100
6.1.2 Conclusion 102
6.2 Optimizing GPU Algorithms 103
6.2.1 Related Work 104
6.2.2 Proposed Algorithm 108
6.2.3 Experimental Results 110
6.2.4 Conclusion 114
6.3 How Does Connected Components Labeling with Decision Trees Perform on GPUs? 115
6.3.1 Introduction 115
6.3.2 Adapting Tree-Based Algorithms to GPUs 116
6.3.3 Comparative Analysis 118
6.3.4 Conclusion 121
6.4 The Block-Based Approach on GPUs 123
6.4.1 Introduction 123
6.4.2 Preliminaries 124
6.4.3 The Komura Equivalence Algorithm 126

6.4.4 The Block-Based Approach 127
6.4.5 Proposed Algorithms 127
6.4.6 Comparative Evaluation 135
6.4.7 Conclusion 143

7 One DAG to Rule Them All **145**
7.1 Introduction 146
7.2 GRAPHGEN 146
7.2.1 Modelling Algorithms with Decision Tables 147
7.2.2 State Prediction 150
7.2.3 From Trees to DRAGs 152
7.3 Three Showcase Applications 153
7.3.1 Connected Components Labeling 154
7.3.2 Image Skeletonization 155
7.3.3 Contours Extraction 161
7.3.4 What About 3D and GPUs? 167
7.4 Conclusion 169

8 Conclusion **171**

Bibliography **175**

Appendix **193**

A List of Publications **193**

B Additional Experimental Results **197**

List of Abbreviations

ASU Average Speed-Up

BACCA Benchmark Another Chain Code Algorithm

BBDT Block Based with Decision Trees (a 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)

CCIT a variation of BBDT algorithm proposed in [1] (a CCL algorithm)

CCL Connected Components Labeling

CNN Convolutional Neural Network

CPU Central Processing Unit

CT Contour Tracing approach (a CCL algorithm)

CTB Configuration-Transition-Based (a CCL algorithm)

CTBE Configuration-Transition-Based Extended/Enhanced (a CCL algorithm)

DT Decision Table

DTree Decision Tree

GCC GNU Compiler Collection

GPU Graphic Processing Unit

GRAPHGEN GRAPH GENerator

KE Komura Equivalence (a GPU-based CCL algorithm)

LBUF Line-Based Union-Find

LE Label Equivalence (a GPU-based CCL algorithm)

LSL Light Speed Labeling (a CCL algorithm)

LUT LookUp Table

ODT Optimal Decision Tree

OS Operating System

PRED optimized state PREDiction (a CCL algorithm)

RemSP interleaved Rem's algorithm with SPlicing (labels solving strategy)

SAUF Scan Array-based Union-Find (a CCL algorithm)

SBLA Stripe-Based Labeling Algorithm (a CCL algorithm)

TDM-GCC Twilight Dragon Media - GNU Compiler Collection (a compiler suite for Microsoft Windows)

THeBE THinning evaluation BEnchmark

TTA interleaved Rem's algorithm with SPlicing (labels solving strategy)

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)

YACCLAB Yet Another Connected Components Labeling Benchmark

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 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 Michele Cancilla, Federico Pollastri, Stefano Allegretti and Laura Canalini 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 months.

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 Danilo and Lorena for nurturing me with passion for knowledge and supporting me throughout my life. Thank you to my girlfriend Elena, for all her love and support. Thank you to my brothers Andrea and Stefano.

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 begun 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 [2, 3, 4, 5], perceiving and elaborating an interpretation of dynamic scenes [6, 7, 8, 9, 10], handwriting analysis and speech recognition [11, 12], surveillance, traffic monitoring and autonomous driving [13, 14, 15, 16], people tracking [17, 18], skeletonization [19], image synthesis [20] and so on. This became possible thanks to the increase of processing capabilities, aided by the fast development of Graphics Processing Units (GPUs), and thanks to the collection of massive amounts of data [16, 17, 21, 22], required during the training of the models.

Nowadays, numerous start-ups and new industrial applications come to life thanks to deep learning. Therefore, important questions come to mind: "Is computer vision and binary image processing without machine learning still worth it?", "How significant is the improvement of these kind of algorithms, both in terms of performance and accuracy, in the *deep learning era*?". As a matter of fact, most of the state-of-the-art solutions on the aforementioned research fields exploit binary image processing algorithms as fundamental pre- or post-processing steps to get to the final results [3, 13, 15, 17, 23] or to prepare training data [20]. When segmenting

images, a highly relevant task in medical imaging [2], *Connected Components Labeling* (CCL) is usually exploited together with voting strategies [24] to remove noise and produce the final segmentation map [3] or to count objects [13]. Thinning, instead, is often used together with contour-tracing, morphological operators, and CCL, whenever a compact representation of the objects inside an image is required [20], as in fingerprint analysis [25], vasculature geometry detection [26, 27], and road mapping [15].

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

Focusing on Connected Component Labeling, this thesis analyzes the state-of-the-art approaches for both sequential CPU-based and parallel CPU- and GPU-based environments, taking care how to fairly measure performance. We then introduce novel approaches to further optimize and improve state-of-the-art of such a kind of algorithms on both sequential and parallel environments, 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.

[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.

To prove the generality of GRAPHGEN, the presented benchmarks are not limited to 2D sequential image processing algorithms, but also include 3D and parallel GPU-based scenarios.

All the source-code developed during my PhD is available on GitHub: 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 currently on going. The DeepHealth project will provide HPC computing power at the service of biomedical applications; and apply 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.

- *H2020 - HIPPOCRATES:* a project submitted under the call SC1-FA-DTS-2018-2020 and currently under review. The HIPPOCRATES project aims at supporting early detection, diagnosis and treatment of several types of cancer through the design and deployment of a full ecosystem for research, development, validation and exploitation of AI-based solutions.

Teaching Activities

- Tutor for the "Fundamentals of Computer Science II" undergraduate course, at Università degli Studi di Modena e Reggio Emilia (2017-2019).

- Lecturer for the “Data Lab” courses, funded by Emilia Romagna and the European Social Fund and organized by CIS —Reggio Emilia—, IFOA —Reggio Emilia—, and Nuova Didactica —Modena— (2018-2019).

- Seminars for the “Multimedia Data Processing” graduate course, Prof. Grana, at Università degli Studi di Modena e Reggio Emilia (2018-2019).

- Seminars for the “Computer Vision” graduate course, Prof. Cucchiara, at Università degli Studi di Modena e Reggio Emilia (2018).

- Laboratory lecturer for the graduate short Masters “Machine Learning - Theoretical and Practical Course”, organized by DEMOCENTER at Università degli Studi di Modena e Reggio Emilia in 2018 and 2019.

- Laboratory lecturer for the graduate Master in “Visual Computing and Multimedia Technologies”, at Università degli Studi di Modena e Reggio Emilia (2017).

Grants and Awards

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

- Within the AImagelab research group —Università degli Studi di Modena e Reggio Emilia— and together with the PRHLT research group —Universitat Politècnica de València—, third place (out of 64 research groups) at the international competition on skin lesion classification (2019 ISIC Challenge).

Journals Reviewing

- IEEE Access - The Multidisciplinary Open Access Journal.

- IEEE Transaction on Image Processing.

- Pattern Recognition Journal - Elsevier.

- IET Computer Vision.

- KES Journal.
- Multimedia Tools and Applications Journal.

Reviews verified on *Publons* at `publons.com/a/1528925/`.

Conferences Attended

- International Conference on Image Analysis and Processing - ICIAP, Trento, Italy, 2019.
- International Conference on Computer Analysis of Images and Patterns - CAIP, Salerno, Italy, 2019.
- IEEE International Conference on Image Processing, Applications and Systems - IPAS, Sophia-Antipolis, France 2018.
- International Workshop on Computer Vision - IWCV, Modena, Italy, 2018.
- IEEE International Symposium on Computer-Based Medical Systems - CBMS, Karlstad, Sweden, 2018.
- Italian Research Conference on Digital Libraries - IRCDL, Udine, Italy, 2018.
- International Conference on Computer Vision - ICCV, Venice, Italy, 2017.
- Italian Research Conference on Digital Libraries - IRCDL, Modena, Italy, 2017.
- International Conference on Image Analysis and Processing - ICIAP, Catania, Italy, 2017.
- Advanced Concepts for Intelligent Vision Systems - ACIVS, Lecce, Italy, 2016.

Seminars Attended

- “Computational and Experimental Neuroscience Toward Artificial Intelligence” —Prof. Jonathan Mapelli (UNIMORE)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, April 10, 2018.
- “Deep Learning for Fault Prediction” —Prof. Roberto Paredes Palacios (UPV)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, February 13-15, 2018.
- “Algoritmi Avanzati (Advanced Algorithms)” —Prof. Mauro Leoncini (UNIMORE) and Prof. Manuela Montangero (UNIMORE)— Master Degree in Computer Science, Department of Physics, Informatics and Mathematics (FIM), Università degli Studi di Modena e Reggio Emilia, Modena, Italy, September 25 - December 22, 2017.
- “The Blockchain Technology” —Alket Cecaj— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, December 14, 2017.
- “NVIDIA Architectures” —Dott. Piero Altoè— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, November 29, 2017.
- “Security and Quantum Technologies - Potential Uses and Risks in the Security Area” —Enrico Prati (CNR)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, November 21, 2017.
- “Handwritten Text Recognition” —Prof. Pastor Moisés Gadea (UPV)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, November 7-10, 2017.
- “Underwater Video Datasets and the VIAME Open-Source Framework for Fisheries Stock Assessment” —Dr. Anthony Hoogs (Kitware)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, October 30, 2017.
- “L’Occhio della Macchina (The Eye of the Machine)” —Prof. Simone Arcagni (UNIPA)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, September 29, 2017.
- “Visión Por Computador” —Prof. Costantino Grana (UNIMORE)— Universitat Politècnica de València, Valencia, Spain, May 22-25, 2017.

- “Academic English Workshop” —Dott. Silvia Cavalieri (UNIMORE)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, May 9-23, 2017.
- “Come Tutelare i Risultati della Ricerca (How to Protect Research Results)” —Dott. Valeria Bergonzini (UNIMORE)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, March 8, 2017.
- “Internet Privacy: Towards More Transparency” —Balachander Krishnamurthy (AT&T)— Università degli Studi di Modena e Reggio Emilia, Modena, Italy, November 21, 2016.

Chapter 2

Literature Survey

In this Chapter we briefly report other research approaches which are related to the topics tackled in this thesis. Although the list could be much longer, we choose to limit ourselves to the methods which are either most relevant for the community or more strictly related to the proposed algorithms. The rest of the Chapter is organized following the flow of the thesis: we will firstly review connected components labeling algorithms (Section 2.1) considering both sequential (Section 2.1.1) and parallel paradigms (Section 2.1.2), we will then move to analyzing other kind of algorithms that can be modeled with decision tables, *i.e.* thinning (Section 2.2) and chain-code (Section 2.3).

2.1 Connected Components Labeling

Connected Components Labeling 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. Since CCL has an exact solution, the main difference between algorithms is the execution time. Moreover, given that labeling represents the base step of many real-time applications, it is required to be as fast as possible. Therefore, research in the field moved towards the optimization of the performance of these algorithms in term of execution speed.

2.1.1 Sequential Algorithms

Introduced by Rosenfeld and Pfaltz [30], 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 [31], Video Surveillance [32], Image Segmentation [33, 34, 35], Medical Imaging Applications [3, 35, 36, 37, 38], Document Restoration [39, 40], Graph Analysis [41, 42], and Environmental Applications [43].

Since 1966, many papers showed algorithms to improve the efficiency of CCL, thus contributing to its very long story full of different strategies that can be classified in many different ways. Moreover, given the relevance of the task, this still represents a hot research topic [44, 45, 46, 47, 48, 49, 50].

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

Raster Scan

Raster Scan algorithms scan the image exploiting a mask (see for instance Fig. 2.1) and solve equivalences between labels using different strategies. *Multiscans* approaches [51], 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 [48, 52, 53, 54] 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.

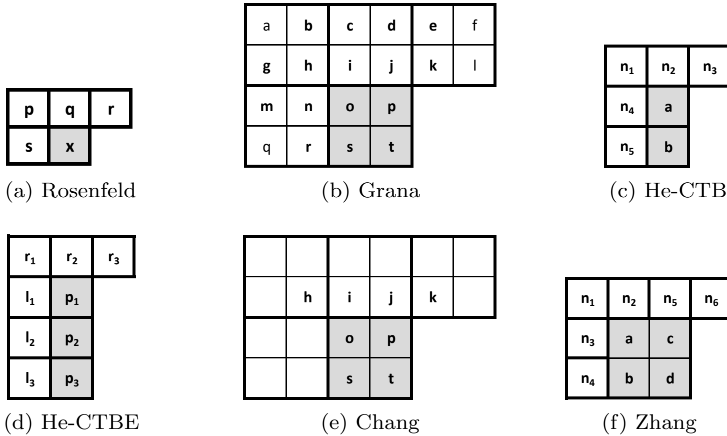
When only statistics about connected components are required (*e.g.* area, perimeter, circularity, centroid), the second scan can be avoided, reducing the total execution time.

Searching and Label Propagation

Searching and Label Propagation algorithms [55] scan the image until an unlabeled pixel is found: it receives a new label which is then repetitively propagated to all connected pixels. The process end when unlabeled pixels no longer exist. These algorithms scan the image in an irregular way.

Contour Tracing

Contour Tracing techniques [56] exploit a single raster scan over the image. During this process all pixels in both the contour and the immediately external background of an object are clockwise tagged in a single operation. Finally, the connected components have to be filled propagating contours' labels. Two scan algorithms have revealed best performances, so our analysis focuses on them.



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

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

Figure 2.1: 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 [30] and since then it has been used by several authors in many papers/algorithms [53, 54, 57]. 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) [48], (d) [58], (e) [1], and (f) [59].

Different solutions allowed performance improvements by avoiding redundant memory accesses [48, 52, 53]. One of the first improvements is the Scan Array-based Union-Find (SAUF) proposed by Wu *et al.* in [53]. This is a reference algorithm because of its very good performance and ease of understanding. The optimization introduced with SAUF reduces the number of neighbors visited during the first scan using a decision tree. The idea is that if two already visited pixels are connected, their labels have already been marked as equivalent in the *union-find* data structure, so we do not even need to check their values.

Since 8-connectivity is usually employed to describe foreground objects, this algorithm was extended in [47, 52] with the introduction of 2×2 blocks, in which all foreground pixels share the same label. In this case, the scanning mask is bigger (Fig. 2.1b), leading to a large number of combinations that produces a complex decision tree, whose construction is much harder. In [60] an optimal strategy to automatically build the decision tree by means of a dynamic programming approach has been proposed and demonstrated. This approach is commonly known as Block Based Decision Tree scanning (BBDT).

He *et al.* [48] were the first to realize that, thanks to the sequential approach taken, when the mask shifts horizontally through the image, it contains some pixels that were already inside the mask in the previous iteration. If those pixels were checked in the previous step, a repeated reading can be avoided. They 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.

2.1.2 Parallel CPU- and GPU-based Algorithms

Parallel GPU-based CCL algorithms can be divided into two disjoint sets, depending on the number of kernel executions. Iterative algorithms repeat one or more kernels until no more changes in the data structures that they modify are detected. Such a situation is called *convergence*, and the number of kernel calls needed to reach it depends on the configuration of the input data. Conversely, direct algorithms are characterized by a fixed number of kernel executions.

Iterative algorithms

Label Equivalence (LE), proposed by Hawick *et al.* [61] in 2010, is an iterative algorithm that records *union-find* trees using an auxiliary data structure. In the first step, both the output image and *union-find* data are initialized with sequential values. The algorithm then consists of three kernels that are repeated in sequence until convergence. They aim at propagating the minimum label through each connected component, exploiting a procedure to flatten *union-find* trees at every step.

In 2011, Kalentev *et al.* [62] proposed an Optimization of Label Equivalence (OLE), noticing that the need for a separate data structure to store label equivalences could be removed by directly using the output image.

Zavalishin *et al.* [63] were the first to apply the block-based strategy proposed by Grana in [52] to a data-parallel CCL algorithm. This approach is based on the observation that, when dealing with 8-connectivity, foreground pixels in a 2×2 block always share the same label. The proposed strategy, which is a variation of Label Equivalence named Block Equivalence (BE), makes use of two additional data structures besides the output image: a block label map and a connectivity map, respectively to contain blocks labels and to record which blocks are connected together. At the beginning of the algorithm, the image is divided into blocks, a label is assigned to each of them, and the necessary information about blocks connectivity is calculated and stored into the connectivity map for future use. The structure of the algorithm is the same as OLE, with the exception that it operates on blocks instead of single pixels. When convergence is met, a final kernel is responsible for copying block labels into pixels of the output image.

Direct algorithms

The first GPU CCL algorithm that makes use of *union-find* was proposed by Oliveira *et al.* in 2010 [64]. We will refer to it as UF. The output image is initialized with sequential values as usual. Then, *union-find* primitives are used to join together the trees of neighbor pixels. Finally, a flattening of trees ends the task. The algorithm is first performed on rectangular tiles, and then large connected components are merged in a subsequent step.

In 2015, Yonehara and Aizawa proposed Line-Based Union-Find [65] (LBUF), a variation of UF that employs single lines as tiles in the first step.

This choice reduces the neighborhood of a pixel to a subset containing only the two neighbors which belong to the same row, allowing to simplify the logic of the local step. The remaining of the algorithm is left unchanged, except for the use of *InlineCompression* to speed up the flattening of trees.

Komura Equivalence (KE) [66] was created in 2015, as an improvement over Label Equivalence. Anyway, it has more in common with Union-Find. Indeed, its structure is very similar to that of UF, but for a different initialization, which starts building *union-find* trees while assigning the initial values to the output image. An improved version of KE, that gets rid of many redundant `Union` operations, has been proposed by Playne *et al.* [67]. The original algorithm and the aforementioned optimization employ 4-connectivity. A 8-connectivity variation has been presented in [68].

Distanceless Label Propagation (DLP) [69] is a proposal that tries to put together positive aspects of both UF and LE. The general structure is similar to that of UF, with the difference that a `Union` is performed between each pixel and the minimum value found in a 2×2 square. The `Union` procedure itself is implemented in an original and recursive manner. A detailed description of the *union-find* approach is available in Section 3.2.

2.2 Image Skeletonization

Thinning is another fundamental algorithm used in many computer vision and image processing tasks, which aims at providing an approximate and compact representation of the elements (objects) inside images. It can be defined as the successive removal of outermost layers of an object until only a skeleton of unit width remains [70]. Firstly introduced in the 1950 as a data compression strategy [71], the thinning procedure is nowadays used as a pre- or post-processing step in many different applications, ranging from medical imaging [26, 27] to handwritten text recognition [72, 73] and fingerprint analysis [25]. Therefore, having an efficient and effective algorithm is extremely important.

In the literature, a lot of approaches to solve the problem have been detailed. The algorithm proposed by Zhang and Suen (ZS) in [74] is one of the most famous and used, given its efficiency and simplicity. This algorithm is based on the 8-neighbor connectivity (Fig.2.2a) and exploits two sub-iterations that are iteratively performed to remove pixels and obtain the final result. In [75], Chen and Hsu (CH) improved the output visual

appearance of the Zhang-Suen approach, by fixing some corner cases, and proposed a LookUp Table (LUT) solution to speed up the process.

Holt *et al.* [76] tackled the problem by a different perspective and proposed an improvement on the Zangh-Suen technique which requires less iterations, at the expense of examining a larger neighborhood (from 3×3 to 4×4). Even though the algorithm solves some of the ZS drawbacks, the need to access more pixels makes it slower, especially when implemented on sequential machines [77].

The algorithm by Guo and Hall [78] allows to better cope with 2×2 squares and diagonal lines inside images using a set of rules that is very similar to the one proposed by Lü and Wang [79].

These solutions have been proposed some decades ago, but are still commonly used [25, 26, 27] and included in many image processing libraries, such as OpenCV.

Given its intrinsically iterative nature the thinning procedure is expensive and usually very slow, especially when applied on high resolution images. Anyway, a lot of approaches have been proposed to improve performances without affecting the output result. Besides the already mentioned LUT technique, an efficient neighborhood exploration technique based on decision trees has been applied on the ZS algorithm [80]. The authors experimentally proved that the use of an optimal Decision Tree (DTree) allows to dramatically reduce the number of memory accesses to be performed in order to explore the neighborhood, thus improving the overall performance of the algorithm, even when compared to implementations based on lookup table.

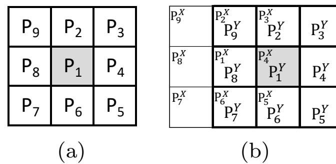


Figure 2.2: Naming convention for the pixel in the neighborhood of P_1 (a) and their overlap when the mask is shifted for processing the next pixel (b).

2.3 Chain-Code

Contours extraction is another common algorithm in binary image processing, usually exploited to represent objects shapes. It has a key role in image storage and transmission, but it is also important for what concerns shape recognition and shape analysis in pattern recognition. In a binary image, a contour is a sequence of foreground pixels that separates an object (connected component) from the background. The output of a contour extraction algorithm is a set of contours, that can be represented in several ways. The simplest is a list of coordinates, but more compact representations also exist: as an example, in Freeman chain-code —the first approach for representing digital curves presented in literature— only one coordinate is stored for a contour, and then all other pixels are identified by a number (from 0 to 7) that encodes its relative position w.r.t. the previous one [81].

In particular, the chain-code scheme defines eight codes in the directions shown in Fig. 2.3b. In this case movements from the center of one pixel to the center of an adjacent one are described. Chain codes having even value have unit length while those having odd value have length $\sqrt{2}$.

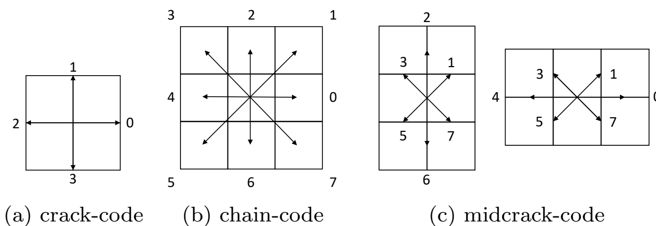
Alternative representation of object outlines have been used in the literature, namely crack-codes and midcrack-codes. The crack code scheme defines four codes in the directions shown in Fig. 2.3a, where each line has length one pixel. The contour is formed by moving on the outer edge of every pixel of the object contour, which means moving along the cracks between adjacent pixels having complementary values. Usually it is assumed that the direction of travel is such that the object pixels are on the right-hand side of the crack. The midcrack-code [82], instead, defines the eight codes shown in Fig. 2.3c. This encoding scheme defines movement from the midpoint of one crack to the midpoint of an adjacent one. Even valued codes have unit length while odd-valued codes have length $1/\sqrt{2}$. In [83] it is shown that both crack and midcrack coded strings may be obtained from the chain-code string of the contour, and the other way around. For this reason and because of its popularity, in the following we will focus on chain-code only.

Since chain-code techniques allows to preserve information while providing a considerable data reduction, many pattern recognition and topology algorithms based on this representation have been proposed in the last decades.

In [84] the author proposes a new algorithm to describe and generate spirals —a recurring pattern in nature (fingerprints, teeth, galaxies and so on) and thus an important topic for computer vision and pattern recognition— by means of the Slope Chain Code, a variation of the Freeman chain-code originally presented in [85]. In [86] a face identification methodology based on chain codes encoded face contours is presented, allowing higher efficiency with respect to normal template matching algorithms for face detection.

Chain-code has also been used for many object recognition tasks [87]: various shape features extraction, contour smoothing and correlation for shape comparison may be obtained directly from this representation [88]. In [89], a feature extraction method using the chain-code representation of fingerprint ridge contours is presented. The representation allows efficient image quality enhancement and detection of fine minutiae feature points. The authors estimate the direction field from a set of selected chain-codes, thus being able to enhance the original fingerprint image using a dynamic filtering scheme that takes advantage of the estimated direction flow of the contours.

In general, contours extraction is applied as one of the first steps of an image processing pipeline, therefore, it should be as fast as possible. Most contours extraction algorithms follow a boundary tracing approach, that hinders the performance causing several cache misses. Anyway, some works for raster scan chain-code extraction algorithms have been proposed in literature [90, 91, 92].



(a) crack-code (b) chain-code (c) midcrack-code

Figure 2.3: Different contour coding schemes: (a) crack-code, (b) chain-code, and (c) midcrack-code.

Chapter 3

Mathematical Preliminaries

This Chapter introduces the basic notations used throughout the thesis. 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*.

3.1 Basic Notations and Definitions

We will call I_2 an image defined over a two dimensional rectangular lattice $\mathcal{L}_2$, and $I_2(p)$ the value of pixel $p \in \mathcal{L}_2$, with $p = (p_x, p_y)$. Two different kinds of neighborhood can be defined (*4-neighborhood* and *8-neighborhood*):

$$\mathcal{N}_4(p) = \{q \in \mathcal{L}_2 \mid |p_x - q_x| + |p_y - q_y| \leq 1\} \tag{3.1}$$

$$\mathcal{N}_8(p) = \{q \in \mathcal{L}_2 \mid \max(|p_x - q_x|, |p_y - q_y|) \leq 1\} \tag{3.2}$$

Two pixels, p and q, are said to be *4-neighbors* if $q \in \mathcal{N}_4(p)$, that implies $p \in \mathcal{N}_4(q)$, and are said to be *8-neighbors* if $q \in \mathcal{N}_8(p)$, that implies $p \in \mathcal{N}_8(q)$. From a visual perspective, if we imagine pixels as square-sized, p and q are *4-neighbors* if they share an edge, and they are *8-neighbors* if they share an edge *or* a vertex.

Similar concepts can be defined for three-dimensional images, also known as *volumes*. Given a volume I_3, defined over a three dimensional lattice $\mathcal{L}_3$, we define two kinds of neighborhood of a voxel (volume pixel) $v \in \mathcal{L}_3$, with $v = (v_x, v_y, v_z)$:

$$\mathcal{N}_6(v) = \{v \in \mathcal{L}_3 \mid |v_x - w_x| + |v_y - w_y| + |v_z - w_z| \leq 1\} \tag{3.3}$$

$$\mathcal{N}_{26}(v) = \{v \in \mathcal{L}_3 \mid \max(|v_x - w_x|, |v_y - w_y|, |v_z - w_z|) \leq 1\} \tag{3.4}$$

This time we can visualize voxels as cubes. So, *6-neighbors* voxels share a side, while *26-neighbors* ones share a side *or* an edge *or* a vertex.

From now on, we will use the word *image* to refer to 3D volumes also, and generic symbols $\mathcal{L}$ and I in definitions suitable for both dimensionalities. In a binary image, meaningful regions are called *foreground* ($\mathcal{F}$), and the rest of the image is the *background* ($\mathcal{B}$). Following a common convention, we will assign value 1 to foreground pixels, and value 0 to background ones:

$$\mathcal{F} = \{p \in \mathcal{L} \mid I(p) = 1\} \tag{3.5}$$

$$\mathcal{B} = \{p \in \mathcal{L} \mid I(p) = 0\} \tag{3.6}$$

The aim of connected components labeling is to identify disjoint objects composed of foreground pixels. So, for a chosen neighborhood definition (simply called $\mathcal{N}$), and given two foreground pixels $p, q \in \mathcal{F}$, the relation of *connectivity* $\diamond$ can be defined as:

$$p \diamond q \Leftrightarrow \exists\{s_i {\in} \mathcal{F} \mid s_1{=}p, s_{n+1}{=}q, s_{i+1}{\in}\mathcal{N}(s_i), i{=}1, \ldots, n\} \tag{3.7}$$

We say that two pixels p, q are *connected* if the condition $p \diamond q$ is true. The above definition means that a path of connected pixels exist, from p to q. Note that, with this formalism, background pixels are excluded from the concept of connectivity. Given that pixel connectivity satisfies the properties of *reflexivity*, *symmetry* and *transitivity*, $\diamond$ is an equivalence relation. Therefore, the equivalence class of a pixel p is denoted as $[p]$ and is defined as the set:

$$[p] = \{q \in \mathcal{F} \mid p \diamond q\} \tag{3.8}$$

In this case, equivalence classes based on $\diamond$ relationship are called *connected components*. Every two connected components $[p]$ and $[q]$ are either equal

or disjoint. Therefore, the set of all connected components is a partition of $\mathcal{F}$.

Connected components labeling algorithms aim at assigning a different label, typically an integer number, to every connected component. When applied to an image I defined over a lattice $\mathcal{L}$, 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. [52].

3.2 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 [93].

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 $\mathcal{P}$.

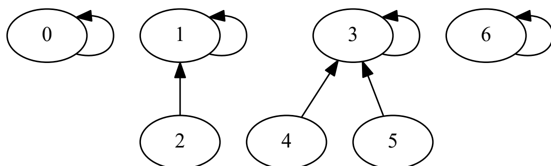
A directed edge leading from b to a, with $id_a < id_b$, states that b and a belong to the same subset. According to the definition of directed rooted tree, we will say that a is the *father* of b.

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

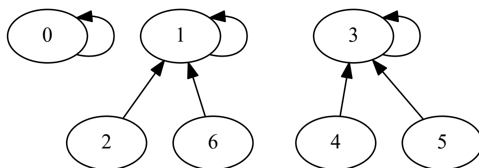
```
1:  function FIND(L, a)
2:      while L[a] ≠ a do
3:          a ← L[a]
4:      return a

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
```

INDEXES →	0	1	2	3	4	5	6
BEFORE UNION(2,6) →	0	1	1	3	3	3	6
AFTER UNION(2,6) →	0	1	1	3	3	3	1

(a)



(b)



(c)

Figure 3.1: 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.

Of course, each element can have at most one out-going 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 father node, or the index itself in the case of roots [53].

Fig. 3.1 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 system call to the driver [94], many GPU CCL algorithms make the *union-find* array coincide with the output labels image L [64, 65, 66, 68, 69, 95].

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 [65].
- `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.* [64], 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* [96, 97]. The algorithm proposed by [53] only adopted the path compression strategy achieving a linear time complexity on average and under certain conditions.

Wu *et al.* [53] call the array that contains the equivalence information the P array (short for parent links). According to [53], 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 [98], 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) [99]. 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 [100].

Chapter 4

Toward Reliable Experiments on Algorithms Performance

The problem of labeling the connected components of a binary image is well-defined and several proposals have been presented in the past. Since an exact solution to the problem exists, algorithms mainly differ on their execution speed. In this Chapter, we propose and describe YACCLAB (Yet Another Connected Components Labeling Benchmark). Together with a rich and varied dataset, YACCLAB contains an open-source platform to test new proposals and to compare them with publicly available competitors, ensuring fairness. Textual and graphical outputs are automatically generated for many kinds of tests, which analyze the methods from different perspectives. An extensive set of experiments among state-of-the-art techniques is also reported and discussed to show the potential of the framework. Additionally, an extension of YACCLAB to deal with 3D volumes and GPU-based algorithms is also presented.

4.1 Introduction

Part of the responsibility for the huge progress in both computer vision and image processing of the recent years may be credited to the broad access to public image and video datasets. Even if datasets have been blamed for narrowing the focus of research on object recognition, reducing it to a single benchmark performance number, it is now clear that the ability to compare different techniques on the same data allows the reader to choose which algorithm suits his needs best [101]. In Computer Science it is not sufficient to reproduce other people tests, but publishing the source code or, at the very least, an executable should be mandatory. Sometimes, just setting the correct parameters may be a problem, changing the final figures by orders of magnitude. This may be referred to as *reproducible research*, *i.e.* an approach at presenting scientific claims together with all information needed to reproduce the presented results, so that others may verify the findings and build upon them.

Benchmarking may be a problem by itself, because measuring performance may not be obvious, but there are some specific tasks in image processing in which the expected result is known. This reduces the burden of the evaluator, since, after checking that the result is correct, the main question left is to measure how fast an algorithm is.

The problem of labeling the connected components of a binary image is such a problem, so one would expect every paper on the subject to focus on the same evaluation method and data. This is not the case. In recent years, many novel proposals have been published and almost none of them compared on the same data [1, 48, 102].

This Chapter tackles the problem of evaluating the speed of execution of different strategies to solve the CCL problem on binary images.

There are three aspects to keep into account when measuring the "speed of execution" of a family of algorithms: the data on which the algorithms are tested, the hardware capabilities and the quality of the software implementation. Purists may horrify at our omission of computational complexity, but the fact is that CCL is inherently a linear algorithm if we separate it from the equivalence solving (the *union-find* problem), whose computational complexity has been already well studied in depth [97].

The base step for all comparisons is to work on the same data, so our contribution is to provide a public dataset of binary images without any license limitations, or synthetically generated ones. We tried to cover

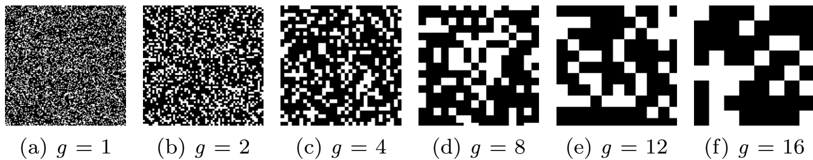
different application scenarios for CCL algorithms such as motion analysis, document processing, OCR, medical imaging, and fingerprint analysis.

Not all computer architectures deliver the same performance on all algorithms and this may also be true for CCL ones. Moreover, the compiler used may significantly impact on the performance of algorithms. The solution we propose to figure out these problems is to provide an open-source *C++* project with a very permissive license, in order to let anyone take the provided algorithms and test them on his own setting, verifying any claim found in the literature (ours included).

Everyone is well aware that software optimization is not an easy task, so the quality of software implementation may change a wonderful algorithm in an unusable junk. Unfortunately, providing source code is not a common requirement for papers to be published, and we had to re-implement many algorithms for which the source code was not available. A positive note is that, being our project open-source, any author believing we did him wrong is welcome to provide a better implementation.

The evaluation framework[1] is called Yet Another Connected Components Labeling Benchmark (YACCLAB in short), and the accompanying dataset is the YACCLAB Dataset.

In the following, we will describe the dataset (Section 4.2), provide some details on how the framework works and how to extend it (Section 4.3), and summarize the CPU-based algorithms currently available in YACCLAB (Section 4.4). An extensive set of experiments is reported in Section 4.5, discussing and motivating the results. Section 4.6 describes how the framework can be extended to GPU and 3D CCL algorithms. Finally, in Section 4.7, we draw some conclusions.



(a) $g = 1$ (b) $g = 2$ (c) $g = 4$ (d) $g = 8$ (e) $g = 12$ (f) $g = 16$

Figure 4.1: Sample images taken from the granularity dataset: all images have a foreground density of 30% and a granularity varying from 1 to 16.

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

4.2 The Dataset

Following a common practice in the literature, we built a dataset that includes both synthetic and real images. The provided dataset is suitable for a wide range of applications, ranging from document processing to surveillance, and features a significant variability in terms of resolution, image density, variance of density, and number of components (see Table 4.1 as a reference). All images are provided in 1 bit per pixel PNG format, with 0 (black) being background and 1 (white) being foreground. The dataset can be automatically downloaded during the set up of the YACCLAB project or it can be found in [103].

Synthetic Images. Random noise images have been used in many papers to evaluate CCL results [52, 98, 104, 105] because connected components in such images have complicated geometrical shapes and complex connectivity to be analyzed. For this reason, we included in our dataset two different set of synthetic images:

1. *Classical:* is the collection of images publicly available in [52], being this the only one already published and used in several other works [48, 106, 107]. These images contain black and white random noise with nine different foreground densities (from 10% up to 90%), and with resolutions ranging from 32×32 to a maximum of 4096×4096 pixels. For every combination of size and density, ten images are provided, making a total of 720 different images. The resulting dataset allows to evaluate performance in terms of scalability on the number of pixels and on the number of labels, which is somehow related to density. For the sake of completeness, this set of images have been generated through the Pseudo Random Number Generator (PRNG) of the *C* standard library implemented in Visual Studio 2008.

2. *Granularity:* encompasses a set of black and white random noise images generated as described by Cabaret *et al.* in [102]. This dataset allows to test algorithms varying not only the pixels density but also their granularity g (*i.e.*, dimension of minimum foreground block), underlying the behaviour of different proposals when the number of provisional labels changes. All the images have a resolution of 2048×2048 and are generated with the Mersenne Twister MT19937

[108] random number generator implemented in the *C++* standard and starting with a *seed* equal to zero. Density of the images ranges from 0% to 100% with step of 1% and for every density value 16 images with pixels blocks of $g \times g$ with $g \in [1, 16]$ are generated. Moreover, the procedure has been repeated 10 times for every couple of density-granularity for a total of 16 160 images.

Natural Images. The second set of images we include is the Otsu-binarized [109] version of the MIRflickr dataset [110], publicly available under a Creative Commons License. It contains 25 000 standard resolution images taken from Flickr, with an average resolution of 0.18 megapixels. There are few connected components (492 on average) and simple patterns, so the labeling is quite easy and fast. Images have an average density of 0.4459 foreground pixels, with a variance of 0.0021. This subset serves again as a comparison with already published results [52, 111].

Medical Images. Another important task where CCL is an indispensable pre-processing operation is medical image analysis. This dataset, provided by Dong *et al.* [112], is composed of histological images and allows us to cover this fundamental field. Dong *et al.* provide both original images and a binarized version of those images, which are included in the YACCLAB dataset. The process used for nuclei segmentation and binarization is described in [112]. The resulting dataset is a collection of 343 binary histological images with an average amount of 1.21 million pixels to analyze and 484 components to label.

Table 4.1: YACCLAB 2D-datasets details.

	Mpix	*Density*	*Variance*	*#CC*
3DPeS	0.41	0.0263	0.0005	320
Fingerprints	0.14	0.2380	0.0084	809
Hamlet	2.71	0.0789	0.0003	1477
Tobacco800	4.60	0.0475	0.0021	2107
XDOCS	16.49	0.0918	0.0005	15 282
Medical	1.21	0.2469	0.0062	484
MIRflickr	0.18	0.4459	0.0365	492
Classical	2.80	0.4996	0.0669	63 107
Granularity	4.19	0.5000	0.0850	8725

Document Images. CCL is one of the initial pre-processing steps in most layout analysis or OCR algorithms. Therefore, to cover for Document Analysis applications, three more datasets are added:

1. *Hamlet:* this is a set of 104 images scanned from a version of the Hamlet found on Project Gutenberg[2]. Images have an average amount of 2.71 million of pixels to analyze and 1447 components to label, with an average foreground density of 0.0789.

2. *Tobacco800:* it is composed of 1290 document images and it is a realistic database for document image analysis research, as these documents were collected and scanned using a wide variety of equipment over time. Resolution of documents in Tobacco800 varies significantly from 150 to 300 dpi and the dimensions of images range from 1200 by 1600 up to 2500 by 3200 pixels [113].

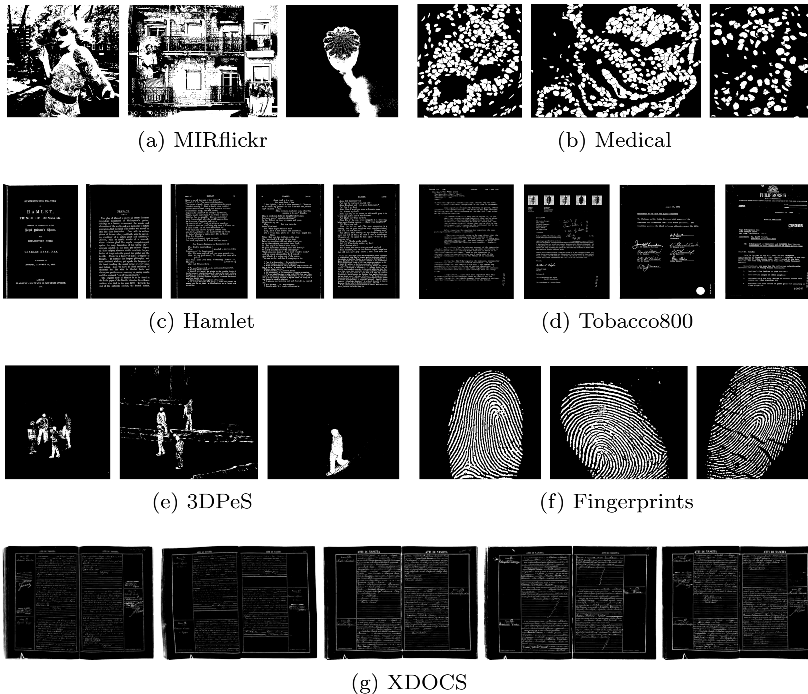
3. *XDOCS:* this is a collection of high resolution historical document images taken from the large number of civil registries that are available since the constitution of the Italian state [39, 114, 115]. XDOCS is composed of 1677 images with an average size of 4853×3387 and 15 282 components to analyze. As for most of document dataset, it has a low foreground density of 0.0918.

Fingerprints Images. This dataset counts 960 fingerprint images collected by using low-cost optical sensors or synthetically generated. These images were taken from three fingerprint verification competitions (FCV2000, FCV2002 and FCV20040 [116]). In order to fit those images for a CCL application, fingerprints have been binarized using an adaptive threshold [117] and then negated in order to have foreground pixels with value 1. Most of the original images have a resolution of 500 dpi and their dimensions range from 240×320 up to 640×480 pixels.

Surveillance Images. The final set of images included in YACCLAB comes from a surveillance dataset, namely 3DPeS (3D People Surveillance Dataset [118]). 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

[2] `http://www.gutenberg.org`

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 [109]. The analysis of the foreground masks to remove small connected components and to match the nearest neighbors is a common application for CCL.



(a) MIRflickr (b) Medical

(c) Hamlet (d) Tobacco800

(e) 3DPeS (f) Fingerprints

(g) XDOCS

Figure 4.2: Sample images from the YACCLAB real datasets.

4.3 The Project

YACCLAB is an open-source project that enables researchers to test CCL algorithms under extremely variable points of view. The software requires the OpenCV 3.0 library (or higher) to work. A configuration file placed in

Table 4.2: YAML configuration file of the YACCLAB project.

	Parameter Name	Description
General	perform	Dictionary which specifies the kind of tests to perform (correctness, average, average_ws, density and size, granularity and memory).
	correctness_tests	Dictionary indicating the kind of correctness tests to perform.
	tests_number	Dictionary which sets the number of runs for each test available.
	algorithms	List of algorithms on which apply the chosen tests.
Datasets	check_datasets	List of datasets on which CCL algorithms should be checked.
	average_datasets	List of datasets on which average test should be run.
	average_ws_datasets	List of datasets on which average_ws test should be run.
	memory_datasets	List of datasets on which memory test should be run.
Utilities	paths	Dictionary with both input (datasets) and output (results) paths.
	write_n_labels	Whether to report the number of connected components in the output files.
	color_labels	Whether to output a colored version of labeled images during tests.
	save_middle_tests	Dictionary specifying, separately for every test, whether to save the output of single runs, or only a summary of the whole test.

the installation folder allows the user to specify which kind of tests should be performed, on which datasets, and on which algorithms. A complete description of all configuration parameters is reported in Table 4.2.

The benchmark provides the following tests, which are deeply described in the following of this Section: *correctness*, average run-time, also called *average* in the following of this thesis, average run-time with steps, which will be referred to as *average_ws*, *density*, *size* and *granularity* tests on synthetic images, memory accesses or simply *memory* test.

It is important to highlight that each test (except correctness and memory ones) is repeated more times per image as specified by the configuration parameter `tests_number`: the minimum execution time for each image is then considered. The use of minimum is justified by the fact that, in theory, an algorithm on a specific environment will always require the same time to execute. This time was computable in exact way on non multitasking single core processors (8086, 80286). Nowadays, too many unpredictable things are happening in the background, independently with respect to the specific algorithm. Anyway, an algorithm cannot use less than the required clock cycles, so the best way to get the "real" execution time is to use the minimum value over multiple runs. The probability of having a higher execution time is then equal for all algorithms. For that reason, taking the minimum is the only way to get reproducible and stable

```
1  class Labeling {
2  public:
3      static cv::Mat1b img_;
4      cv::Mat1i img_labels_;
5      unsigned n_labels_;
6      PerformanceEvaluator perf_;
7
8      Labeling() {}
9      virtual ~Labeling() = default;
10
11     virtual void PerformLabeling();
12     virtual void PerformLabelingWithSteps();
13     virtual void PerformLabelingMem(std::vector<unsigned long>& accesses);
14
15     virtual void FreeLabelingData() { img_labels_.release(); }
16 };
```

Listing 4.1: Labeling base class from which CCL algorithms have to inherit.

results from one execution of the benchmark to another on the same environment. Two other strategies useful to obtain stable execution times are: *(i)* stopping all the background processes and *(ii)* disabling page swapping during the execution of the benchmark.

YACCLAB has been designed with extensibility in mind, so that new resources can be easily integrated into the project. To introduce a new CCL algorithm into the benchmarking system, it must be compliant with a base interface (Listing 4.1) implementing the following methods:

- *PerformLabeling*: includes the whole code of the algorithm and it is necessary to perform average, density, size and granularity tests;

- *PerformLabelingWithSteps*: implements the algorithm, dividing it in steps (*i.e. alloc/dealloc*, *first_scan* and *second_scan* for those which have two scans, or *all_scan* for the others) in order to evaluate every step separately;

- *PerformLabelingMem*: is an implementation of the algorithm that traces the number of memory accesses whenever they occur.

The *C++* savvy will notice the fact that the labeling methods are declared virtual, thus adding an overhead to the function call. We measured the impact of this and verified that it is many orders of magnitude lower than the time required by the algorithms, being in fact negligible. Additionally, it is the same for all algorithms.

All CCL algorithms included in YACCLAB implement the 8-connectivity. Moreover, in order to compare different proposals as fairly as possible, we standardized shared code preferring, for example, the *new* statement to allocate memory or using, when possible, the same data types.

We look at YACCLAB as a growing effort towards better reproducibility of CCL algorithms, so implementations of new and existing labeling methods are welcome.

Correctness Test. The first kind of test that YACCLAB enables is related to correctness. In order to check the correctness of an implementation, indeed, the output of an algorithm is compared with that of the Scan plus Array-based Union-Find algorithm [54], which is assumed to be a correct reference point. Before making the comparison, labels indexes are changed to force a row major ordering: different labeling paradigms

may assign different labels to the same object and this is not considered an error. The datasets on which correctness test shall be executed have to be specified through the `check_datasets` list in the configuration file. An additional dataset to the ones already described in Section 4.2 has been specifically designed for correctness tests. This collection, called *check* and included in YACCLAB, contains a set of 20 binary images with special sizes (*i.e.* odd number of rows/columns or single row/column) and a chessboard texture to test algorithms in the most critical cases.

Average Run-Time Test. Average run-time test executes algorithms on every image of a specified dataset and reports the average execution times in three different formats: plain text files, histogram charts, either in color or in gray-scale, and LaTeX tables, which can be directly included in research papers.

Such kind of test can be applied on all YACCLAB datasets and the `average_datasets` list enables the user to specify which ones should be selected. YACCLAB performs average test only whether the associated entry in the `perform` dictionary of the configuration file is set to "true".

It should be noted that the execution times calculated by average test always include memory allocation. We strongly believe that excluding memory allocation from the execution time is not impartial. Indeed, each algorithm may require a different number of tables and obviously these impact on performance. In real applications, CCL is applied on images of different sizes and this requires to reallocate data when needed. For this reason, it is mandatory to include the memory allocation time to evaluate the performance of an algorithm. On the other hand, there are cases in which it could be fair to compare algorithms without considering memory allocation: in an embedded system in which images are always captured with the same size, for example, could be realistic to allocate memory only once. In order to cover these circumstances, we introduced the *average_ws* test in our benchmarking system.

Average Run-Time Test with Steps. This test evaluates the performance of an algorithm separating the allocation/deallocation time (*alloc/dealloc*) from the time required to compute labeling. Moreover, if an algorithm employs multiple scans to produce the correct output labels, YACCLAB will store the time of every scan and will display them separately.

To understand how YACCLAB computes the memory allocation time for an algorithm on a reference image, it is important to underline the subtleties involved in the allocation process. Indeed, all modern operating systems (not real-time, nor embedded ones, but certainly Windows and Unix) handle virtual memory exploiting a *demand paging* technique, *i.e.* demand paging with no pre-paging for most of Unix OS and cluster demand paging for Windows OS. This means that a disk page is copied into physical memory only when it is accessed by a process the first time, and not when the allocation function is called. Therefore, it is not possible to calculate the exact allocation time required by an algorithm, which computes CCL on a reference image, but its upper bound can be estimated using the following approach:

1. forcing the allocation of the entire memory by reserving it (*malloc*), filling it with zeros (*memset*), and tracing the time;
2. calculating the time required by the assignment operation (*memset*), and subtracting it from the one obtained at 1;
3. repeating the previous points for all data structures needed by an algorithm and summing times together.

This will produce an upper bound of the allocation time because caches may reduce the second assignment operation, increasing the estimated allocation time. Moreover, in real cases, CCL algorithms may reserve more memory than they really need, but the *demand paging*, differently from our measuring system, will allocate only the accessed pages.

Synthetic Test. Test on synthetic images are useful to evaluate the performance of different approaches in term of scalability on the number of pixels and labels. Moreover, synthetic tests give us the possibility to highlight the behavior of CCL algorithms when the foreground pixels density changes. YACCLAB divides this test into two groups:

- *density and size*: are performed on *classical* dataset and estimate the performance of different CCL algorithms when they are executed on images with varying foreground density and size. The density test calculates the mean execution time of each algorithm over images whose density ranges from 10% up to 90%, with a 10% step. On

the other hand, size test reports average execution time on images having resolutions ranging from 32×32 up to 4096×4096;

- *granularity*: evaluates an algorithm varying density (from 1% to 100%, using a 1% step) and pixels granularity, but not images resolution. This test was proposed in [102] and its inclusion in YACCLAB allows to also verify the performance claims of CCL algorithms on this demanding case. The output results display the average execution time over images with the same density and granularity.

Memory Accesses Test. This test is useful to correlate the performance of an algorithm to the number of memory accesses. The memory test computes the average number of accesses to the label image (*i.e.* the images usually used to store the provisional and then the final labels for the connected components), the average number of accesses to the binary image to be labeled and, finally, the average number of accesses to data structures used to solve the equivalences between label classes. Moreover, if an algorithm requires extra data, memory test summarize them as "other" accesses and returns the average. Furthermore, all contributions of each algorithm and dataset are summed together in order to show the total amount. Since counting the number of memory accesses imposes additional computations, the code implementing this test is not shared with that implementing the others.

4.3.1 NULL labeling

The *NULL labeling* is a new "algorithm" which defines a lower bound limit for the execution time of CCL on a given machine and dataset. As the name suggests, the *NULL labeling* does not provide the correct connected components for a given image. It only copies the pixels from the input image to the output one. It is important to underline that the lower bound limit defined by the *NULL labeling* can not be overtaken by any CCL algorithm. The operations performed by NULL labeling allow to identify the minimum time required for allocating

Algorithm 2 NULL labeling algorithm. IN and OUT are respectively the input binary image and the output integer image, both of width w and height h

1: **for** $r \leftarrow 0$ **to** $h-1$ **do**
2:     **for** $c \leftarrow 0$ **to** $w-1$ **do**
3:         $OUT(r,c) = IN(r,c)$

the memory of the output image, reading the input image and writing the output one. In this way, all the algorithms may be compared in terms of how costly are the additional operations required.

4.3.2 *Union-Find* Templating

Following the idea of extensibility, an algorithm can also be templated on the *union-find* strategy. YACCLAB is able to compare each algorithm (but those for which the labels solver is built-in) with four different labels solving strategies: standard *Union-Find* (UF), Union-Find with Path Compression (UFPC) [53], Interleaved Rem's algorithm with splicing (RemSP) [99] and Three Table Array (TTA) proposed in [57]. This flexibility allows YACCLAB to better analyze the behavior of a specific algorithm and the user to choose the best combination for his machine, compiler, and operating system. The standardization of the *union-find* algorithms reduces the code variability and provides fairer comparisons. Particular care was used to check that the flexibility introduced with *union-find* templating did not impact on execution time. We compared the produced code at assembly level and the output is the same with and without templates.

4.4 Available Algorithms

Since version 3.0, OpenCV included CCL features. The algorithm implemented was the one described in [53, 54], which is basically equivalent to the one in [104][3]. It uses a pixel based scanning with online equivalence solving by means of a *union-find* technique with path compression, plus a decision tree for accessing only the minimum number of already labeled pixels.

Still, this is the reference algorithm implemented in YACCLAB, because of its very good performance and ease of understanding.

In [52] we proved that different versions of the decision tree are equivalent to the previous one and extended it to Block Based scanning, that is scanning the image in 2×2 blocks. Building the decision tree for that case is much harder, because of the large number of possible combinations. In [60] we proposed a proved optimal strategy to build the decision tree

[3] After the appearance of the YACCLAB project results, OpenCV changed its default algorithm to the fastest one reported in YACCLAB.

by means of a dynamic programming approach. In YACCLAB we provide an implementation of this algorithm which employs the usual OpenCV optimizations on image accesses.

Another variation of Block Based analysis was proposed in [106], which is reported to be faster than the previous one. We also include this algorithm thanks to the availability of the source code on the web pages of authors, making the signature compliant with our standards and adding some checks to deal with images with an odd number of rows and columns as the other algorithms do.

In [119] a different take on labeling, called Light Speed Labeling, was proposed. The paper has a well described pseudocode for the algorithm, and in the journal version [102] further analyses have been performed, also proposing some variations and stating that it is the fastest algorithm available. To our knowledge, no public implementation exists, so we are the first ones to really make it available. This is probably due to two small mistakes in the pseudocode of [105]: a w was called n because of a change in notation between the two papers and a "step 2" was missing in a "for" loop. We implemented both the standard version —this is the name used by the authors—, the compressed version, and the *zero-offset* optimization, that is reported to provide a speed-up of up to 5%.

In order to demonstrate the importance of the algorithm used to solve label equivalences, we include an implementation of [120] which uses a two scan procedure with an online labels solver algorithm (exploiting an array-based structure to store the label equivalences). This technique requires multiple searches over the array at every *union* operation, leading to a clear non-linear behavior with respect to the number of labels.

As a representative of the contour tracing type of algorithms we include the approach proposed in [56]. This approach clockwise tags all pixels in both the contour and the immediately external background area in a single operation. Then, during the raster scan, when an untagged boundary is found, a counterclockwise contour tracing is performed for internal contours. This technique proved to have a linear complexity with respect to the number of labels and run quite fast, also because the filling of the connected components (label propagation after contour following) is cache-friendly for images stored in a raster scan order.

In order to cover a new recent paradigm for CCL, YACCLAB includes two more algorithms: Configuration-Transition-Based [48] originally proposed by He *et al.* in 2014 and Optimized Pixel/State Prediction [121]

proposed by Grana *et al.* in 2016.

The algorithm from He —using a reduced version of the 2×2 scan mask, from the block-based algorithm, to just 1×2 pixels— scans the given image at alternate lines and processes pixels two by two. The authors also define nine different configurations and nine groups of associated actions for the pixels in the current scan mask. Then, in the labeling approach, the next-case decisions are represented as a configuration-transition diagram to obtain better speed. The current *state* of the algorithm is defined by the values of pixels already checked in the current scan mask, and by previous actions executed. This approach allows He *et al.* to save the number of accesses to pixels and to speed-up the labeling process. The design of the algorithm is specific for the CCL problem and there is no prevision of extending it to any other similar task.

In Optimized Pixel Prediction (PRED), instead, we proposed a general paradigm to exploit already seen pixels during the scan phase, in order to minimize the number of times a pixel is accessed. As shown in literature, the decision table which rules the scan step can be conveniently converted to an optimal binary decision tree [52], in which internal nodes represent conditions on mask pixels, and leaves represent actions to be performed on the current pixel of the provisional labels image. Usually, the same decision tree is traversed for each pixel of the input image, without exploiting values seen in the previous iteration, which, if considered, would result in a simplification of the decision tree for that pixel. To go beyond this limitation, we computed a reduced decision tree for each possible set of known pixels: these reduced decision trees are then connected into a single graph, which rules the execution of the CCL algorithm on the whole image. Each leaf of a tree, which represents the action to be performed on a pixel, is connected to the root of a second tree, which should be executed for the next pixel. The obtained graph can then be directly converted into running code. A complete description of PRED is provided in Section 5.1.

Finally, we also include the stripe-based algorithm proposed by Zhao *et al.* [111]. Here, each pair of consecutive rows in the image is taken as a work-region (called *stripe*). Foreground pixels in each stripe are replaced with a global position value (the index of the leftmost pixel of the stripe connected to that pixel). Every stripe is then considered as a rooted tree, and the image is abstracted as the forest composed of all stripes. Stripes are therefore merged together by merging multiple rooted trees. Finally, the image is traversed to find the roots of non-zero pixels and obtain label

values, which are then assigned to the non-zero pixels. In the original paper, the number of rows in the image is supposed to be even, and authors state that no extra auxiliary memory is required. We notice that to store global position values on images having more than 256 labels, pixels should be converted to a larger data type, thus requiring additional memory. To make a fair comparison, our implementation copies the input image to one with 32 bits per pixel, and adds proper checks to deal with the last row.

4.5 Experimental Results

In this Section, experimental results obtained with YACCLAB over the aforementioned algorithms are presented and analyzed.

There are many variables that may significantly influence the performance of an algorithm: the chosen labels solver, the adopted compiler, the operating system on which tests are performed, the machine architecture and last but not least the data on which algorithms are executed. Unfortunately, all these combinations generate too much data to be analyzed and reported here, so we select the most significant and general ones, highlighting the strengths and weaknesses of the state-of-the-art algorithms.

Specifically, we picked nine different algorithms: five of them can use four different labels solving algorithms, one (Light Speed Labeling) has four variations and each can use two labels solving algorithms, for a total of 31 combinations plus the NULL labeling. We have seven datasets for the average run-time test, for a total of 224 values, which we fit in a single results table. We considered two compilers (MSVC 19.11.25508.2 with 02 speed optimizations enabled and GCC 5.1.0 with 03 optimization flag) and two operating systems (Windows 10.0.15063 64 bit and Linux 4.10.0-33-generic 64 bit), for a total of three possible combinations (the Microsoft compiler cannot run under Linux). We selected a single machine in order to showcase the performance of the different algorithms: an Intel Core i7-4770 CPU @ 3,40 GHz (with 4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache), and with 24 GB of RAM available. We understand that a comparison in performance between different machines would be useful, but it would require to replicate all the considerations, leaving anyway space to the question whether the observations on the two selected machines could be applied also to another one. In the following, we will use acronyms to refer to the available algorithms: CT is the Contour

Tracing approach by Fu Chang *et al.* [56], CCIT is the algorithm by Wan-Yu Chang *et al.* [106], DiStefano is the algorithm in [120], BBDT[4] is the Block Based with Decision Trees algorithm by Grana *et al.* [52], SAUF[4] is the Scan Array-based Union-Find algorithm by Wu *et al.* [54], CTB is the Configuration-Transition-Based algorithm by He *et al.* [48], SBLA is the stripe-based algorithm by Zhao *et al.* [111], and PRED is the Optimized Pixel Prediction by Grana *et al.* [121]. Additionally, the four different versions of the Light Speed Labeling algorithm by Lacassagne *et al.* [102] are identified as LSL_STD, LSL_STDZ, LSL_RLE, and LSL_RLEZ, where STD refers to the standard version of the algorithm, RLE refers to the run length encoding optimization and Z is related to the *zero-offset* addressing optimization as described by the authors in the paper. Moreover, NULL is the lower bound limit described in Section 4.3.1. Finally, to identify the labels solver adopted to test an algorithm, the acronyms already presented in Section 4.3.2 are placed after its name.

Average Run-Time Test. Results of average tests are summarized in Tables 4.3 (at the end of this Section), B.1, and B.2 (in Appendix B).

The first conclusion we can draw is that Linux is faster for this task. Extremely so. In particular, when memory allocation becomes important, *i.e.* when the dataset has large images, or the algorithm requires larger data structures, the time required by Windows may be double than Linux. Other tests reported the cause to be a higher allocation time, due to the virtual page commits.

In order to evaluate how labels solver algorithms affect the performance, we estimated the time required by all of them for every algorithm and dataset. Changing the labels solver can lead to significant enhancements for specific combinations of algorithm, data, and operating system, or it can make no difference for others. It appears that the best strategy the user can follow is to test the algorithms on his specific configuration and pick the one which delivers the best performance. For what concerns labels solver algorithms, the following tentative conclusions can be drawn:

- Windows MSCV: RemSP provides the best performance with all algorithms, except with CCIT that uses TTA for best results.

[4]SAUF and BBDT are the algorithms currently included in the OpenCV library (since version 3.2).

- Windows GCC: UF provides the best performance with all algorithms, or it is equivalent to RemSP with CCIT.
- Linux GCC: TTA and RemSP are almost equivalent, and the others are not too different, except, again, with CCIT that significantly favors TTA.

The different memory management, between the two OS, can again be a possible explanation of performance variance between labels solvers. Linux seems to have a better performance, since it allocates memory pages with a speed that doubles the Windows memory allocation capability. TTA is, at least in theory, really efficient when merges are encountered but, unfortunately, this efficiency requires more memory. Therefore, the advantages appear significant under Linux, instead, with Windows, they are frustrated by the allocation costs.

For all we said, expressing a judgment on which algorithm is "best" is extremely difficult, and maybe plain wrong. Under Windows, BBDT has best performance, irrespective of the compiler.

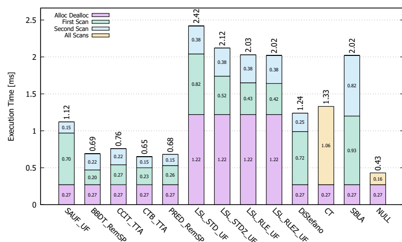
Under Linux, CTB demonstrates, in most cases, the best behavior. This is true for our test machine, and can be justified later by observing the behavior of other tests, but we cannot say which algorithm of the two is the fastest.

For what regards LSL, our tests do not confirm the results on CCL presented in [102], but it is clear that the *zero-offset* optimization is extremely beneficial for LSL_STD, less so for LSL_RLE. The RLE version is most of the times faster than the STD one, even after the *zero-offset* optimization.

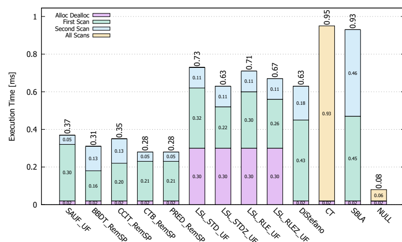
Average Run-Time Test with Steps. In order to discuss the single phases of each algorithm, we need to shorten the algorithm lists. We thus select, for each algorithm and dataset combination, the labels solver which has the lowest total execution time when using the *PerformLabelingWithSteps* methods. This also allows to show the charts produced by YACCLAB in Fig. 4.3 and Fig. B.1 (in Appendix B).

The allocation/deallocation time is stable and depends only on the data structures used: CT, SBLA, and NULL do not have any memory requirement in addition to the output image; SAUF, BBDT, CCIT, CTB, and PRED have the additional requirements of the *union-find* structures, which are one (UF, UFPC, RemSP) or three arrays (TTA); DiStefano has

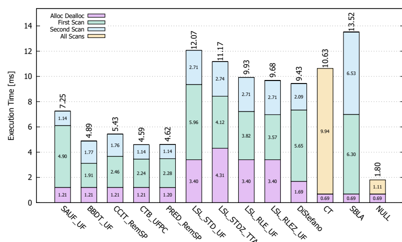
a two arrays structure to handle labels resolution; LSL has always a larger memory footprint. All this is reflected in the time requirements. Note that the time is quantized on the number of virtual memory pages required, so for example on 3DPeS or Fingerprints, there is no difference but for LSL.



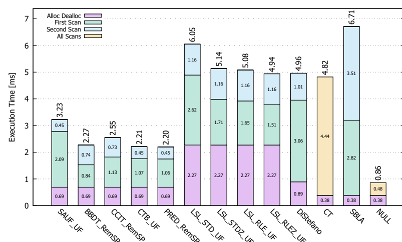
(a) 3DPeS



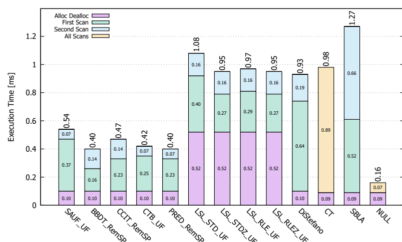
(b) Fingerprints



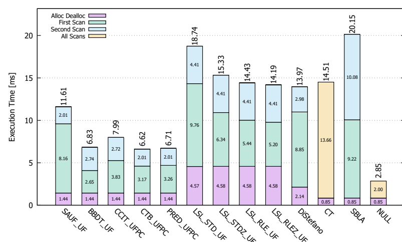
(c) Hamlet



(d) Medical



(e) MIRflickr



(f) Tobacco800

Figure 4.3: Average run-time tests with steps in ms on an Intel Core i7-4770 *CPU* @ 3.40GHz running Linux with GCC 5.1.0. Lower is better.

The first scan of BBDT is always faster than that of the other two scan algorithms, while its second scan is slower. This is clearly due to the fact that the 2×2 mask used requires a bunch of tests in the second scan, which are saved in the first scan. CCIT has exactly the same second scan and the same timings, but its first scan is slower due to a different organization of the decision tree. LSL second scan has to make two indirections and this causes another slowdown. Of course the real problem of LSL is the first scan, which is extremely costly and slower than the other efficient algorithms.

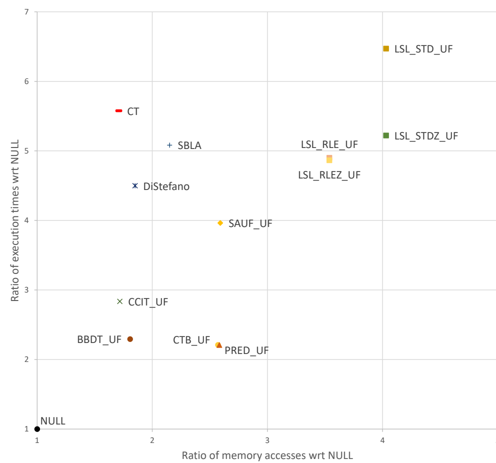


Figure 4.4: Correlation between memory accesses and execution times normalized to the NULL labeling and obtained under Linux with GCC 5.1.0 on Tobacco800 dataset.

The memory savings of SBLA are annihilated by the horrible cache accesses caused by the simultaneous use of the output image as a *union-find* structure. CT is heavily affected by the length of the contours, so its worse performance is obtained on the fingerprints dataset.

Memory Test. YACCLAB can also report the number of memory accesses of each algorithm, for each dataset. We are not reporting these numbers for all combinations, but we simply try to graphically report a summary in Fig. 4.4, which has been computed under Linux, with GCC, on Tobacco800 dataset, using the classical UF algorithm. The chart compares memory accesses and execution time, but normalizes them with reference to the values of NULL labeling. For this reason, the axes start at 1 both for x and y and the values are adimensional.

There is a correlation between number of memory accesses and time, but it is not linear and not perfect. In fact, CT has very few memory accesses, similarly to CCIT, but their access patterns differ a lot: CT is

definitely not cache friendly and thus CCIT is much faster. Furthermore, BBDT uses the same mask of CCIT, requires slightly more accesses, but it is faster than CCIT because the same results are obtained with a more structured and regular code which is easier for the compiler to optimize. CTB and PRED instead use a smaller mask, thus requiring more accesses to both the output image, and the equivalence storage structures. Their mask is anyway used smartly, containing the number of accesses to the input image (they leverage the outcome of one pixel to evaluate the next one), thus allowing for an extremely simple and branchless second scan. As already analyzed before, DiStefano and SBLA have extremely poor labels solving strategy which causes the execution time to be very high. The other algorithms have even higher memory requirements, which cause more slowdowns and impact on performance.

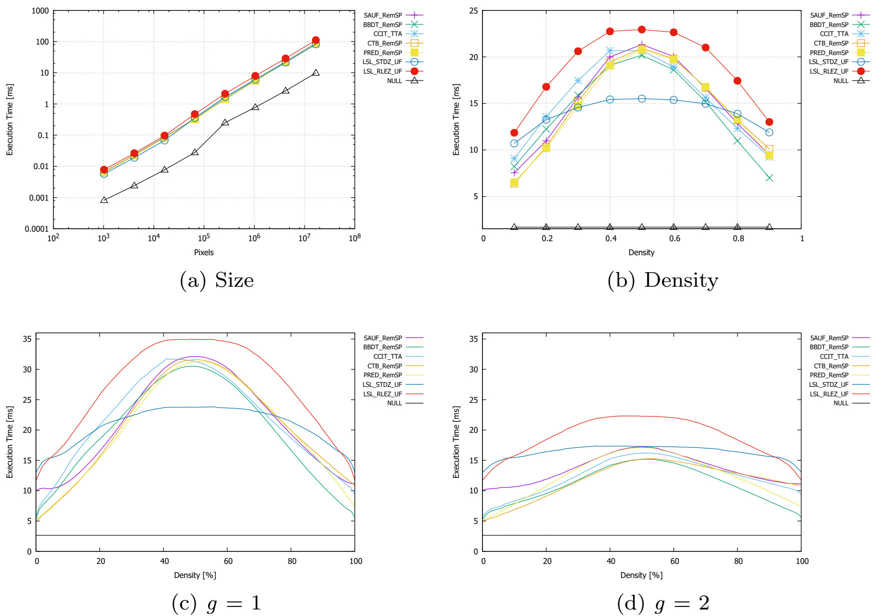
Synthetic Tests. The final results have been obtained on the synthetic datasets and are reported in Fig. 4.5 and Fig. B.2 (in Appendix B). Again, these tests are reported for a reduced set of combinations, selecting the best performing labels solver, under Linux and with GCC. We removed the worse performing algorithms in order to keep the figure readable. The *Density* chart shows how, depending on the number of foreground pixels, the behavior changes. This is mostly related to the errors of the branch prediction unit which heavily affects the algorithms around 0.5. It is interesting to note that PRED and CTB behave extremely similarly and are effective at low densities, LSL_STD is much less affected by the density changes, BBDT is faster at higher densities.

We included these tests in YACCLAB for the sake of completeness, but during the writing of this thesis we realized *why* virtually any conclusion drawn from this is inapplicable to real world images. The problem is that the density of foreground pixels is not the key factor in the branch prediction: the key point is the probability of transition from black to white and viceversa. We are keeping it fixed at 50% using random generation, but in real images this is definitely not the case. So the branch predictor has much less problems and the result are more similar to those observed at lower densities.

Less interesting is the *Size* chart, which shows that the behavior is linear in the number of pixels for all the selected algorithms, as expected. The only point we want to stress is the "jump" observed around 10^5 pixels. This is due to the images not fitting L2 cache anymore and requiring also

L3 cache to be employed to fit the input image.

The *Granularity* charts, on the other hand, highlight the performance of an algorithm when both density of foreground pixels and their granularity change. Of course, when the granularity is equal to 1 (Fig. 4.5c) the algorithms performance have the pattern already observed in the density chart. Then, when the granularity grows, the execution time for middle density images decrease. Again, this can be easily explained considering the branch prediction unit: when pixel blocks in the input image became bigger, the prediction of pixel values, which are totally random, fails less, thus decreasing the cost associated to failures.



(a) Size

(b) Density

(c) $g = 1$

(d) $g = 2$

Figure 4.5: Size (a), Density (b) and Granularity (c), (d) tests on Intel Core i7-4770 *CPU* @ 3.40GHz running Linux with GCC 5.1.0. Some algorithms have been omitted to make the charts more legible. Lower is better.

Table 4.3: Average run-time results in ms obtained under Windows 10 with MSVC 19.11.25508.2 compiler. The bold values represent the best labels solver for a specific CCL algorithm and dataset, the red ones point out the best algorithm for a given dataset. Lower is better.

	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
SAUF_RemSP	0.823	**0.401**	6.206	2.817	**0.506**	9.648	**48.605**
SAUF_TTA	0.830	0.401	6.245	2.849	0.507	9.698	48.959
SAUF_UFPC	**0.823**	0.402	6.212	2.824	0.510	9.657	48.673
SAUF_UF	0.824	0.401	**6.205**	**2.814**	0.510	**9.647**	48.638
BBDT_RemSP	0.650	**0.314**	**5.011**	**2.186**	**0.396**	**7.951**	**40.587**
BBDT_TTA	0.660	0.316	5.038	2.187	0.397	7.995	40.789
BBDT_UFPC	0.651	0.315	5.023	2.203	0.397	7.961	40.684
BBDT_UF	**0.650**	0.317	5.033	2.199	0.397	7.965	40.658
CCIT_RemSP	0.779	0.356	5.898	2.600	0.461	9.389	46.675
CCIT_TTA	**0.741**	**0.348**	**5.666**	**2.520**	**0.451**	**8.923**	**45.271**
CCIT_UFPC	0.741	0.370	5.813	2.634	0.486	9.025	46.227
CCIT_UF	0.763	0.363	5.876	2.598	0.471	9.238	46.633
CTB_RemSP	**0.809**	**0.377**	**6.100**	**2.810**	**0.493**	**9.659**	**48.375**
CTB_TTA	0.941	0.407	7.240	3.184	0.524	11.741	55.597
CTB_UFPC	0.811	0.381	6.111	2.804	0.498	9.643	48.446
CTB_UF	0.892	0.404	7.298	3.147	0.534	11.679	56.029
PRED_RemSP	**0.720**	**0.349**	**5.475**	**2.534**	**0.467**	**8.499**	**44.642**
PRED_TTA	0.730	0.353	5.539	2.603	0.477	8.578	45.736
PRED_UFPC	0.723	0.355	5.496	2.579	0.479	8.528	44.854
PRED_UF	0.723	0.353	5.492	2.573	0.480	8.532	45.408
LSL_STD_TTA	2.149	0.697	16.503	7.136	0.994	27.843	127.648
LSL_STD_UF	**2.139**	**0.694**	**16.466**	**7.122**	**0.991**	**27.793**	**127.433**
LSL_STDZ_TTA	1.716	0.551	13.459	5.802	0.802	22.696	104.932
LSL_STDZ_UF	**1.707**	**0.550**	**13.411**	**5.784**	**0.800**	**22.650**	**104.850**
LSL_RLE_TTA	1.695	0.631	13.722	5.880	0.846	22.600	105.326
LSL_RLE_UF	**1.682**	**0.629**	**13.637**	**5.843**	**0.843**	**22.517**	**104.901**
LSL_RLEZ_TTA	1.767	**0.654**	**14.216**	**6.092**	**0.876**	23.441	**108.819**
LSL_RLEZ_UF	**1.760**	0.654	14.223	6.108	0.878	**23.428**	108.944
DiStefano	0.881	0.606	7.556	4.037	0.850	11.363	90.903
CT	0.988	0.937	9.684	4.517	0.938	12.820	75.670
SBLA	0.878	0.551	7.480	3.559	0.666	10.887	61.673
NULL	0.369	0.097	2.508	1.120	0.165	4.414	21.579

4.6 Support for GPU and 3D CCL Algorithms

In this Section we explain how the YACCLAB benchmark and the associated datasets have been extended to evaluate the algorithms performance also on GPU and on 3D volumes.

In order to reuse the largest possible part of YACCLAB original code, we needed to adapt the function responsible for running tests to a more generic usage. First of all, we designed a wrapper class for input and output data, called `YacclabTensor`, with sub-classes for 2D and 3D images that can reside in host or device memory. The class automatically handles data transfers, so that a copy of the data is always available in the location where it is needed. To be included in YACCLAB, CCL algorithms must be compliant with a base interface (Listing 4.2), implementing specific methods to perform tests.

Polymorphism comes at a certain cost in term of performance. Anyway, we measured its impact and verified that no overhead is introduced in the execution of labeling algorithms. In fact, only framework operations, whose execution time is not critical, have been made generic.

It is important to underline that when it comes to GPU algorithms, in the case that the device has a dedicated memory —which is the most common situation— an important issue to deal with is the need for transferring data from host (CPU) to device (GPU) memory and viceversa. A

```
1  class Labeling {
2  public:
3      std::unique_ptr<YacclabTensorInput> input_;
4      std::unique_ptr<YacclabTensorOutput> output_;
5      PerformanceEvaluator perf_;
6
7      Labeling() {}
8      virtual ~Labeling() = 0;
9
10     virtual void PerformLabeling() {}
11     virtual void PerformLabelingWithSteps() {}
12     virtual void FreeLabelingData() { output_->Release(); }
13 };
```

Listing 4.2: Simplified version of labeling base class. CCL algorithms have to inherit from a specialization of this class. Available specializations are Labeling2D, Labeling3D, GpuLabeling2D, and GpuLabeling3D.

fair comparison between CPU and GPU algorithms would include transfer times in the total execution time. Anyway, some information is necessary about where the input image can be retrieved and where the output of the algorithm is required to be delivered. In fact, in the common case that the input image lies in the host memory and the output is needed there as well, transfer times should be added to the execution time of GPU algorithms. Nevertheless, the opposite can happen as well, *e.g.* if CCL is just a step of a more complex pipeline entirely run in GPU.

Several experimental tests (some of which are reported in the following Chapters) led to the conclusion that state-of-the-art GPU algorithms are only advantageous when the image is already stored in device memory, and the contrary applies to CPU algorithms, because the aforementioned overhead always makes the difference. Thus, we decided to separately compare CPU and GPU algorithms in every performance test.

In order to test 3D algorithms the following datasets of 3D volumes have been introduced to the framework. As for 2D datasets, 3D volumes can be directly downloaded during the YACCLAB setup or they can be found in [122].

OASIS. It is a dataset of medical MRI data already used in the past to evaluate the performance of 3D CCL algorithms [67]. 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 [123]. The dataset consists of 373 volumes, each of which contains 128 slices of 256×256 pixels. To make the images suitable for CCL the original volumes were binarized with the Otsu threshold [109], calculated over the entire volume.

Mitochondria. This is the Electron Microscopy Dataset available in [124] and originally published in [125]. 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 1024×768 pixels.

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 [67]. For these reasons we included

in 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 128×128×128.

Synthetic 3D Volumes. Two datasets of black and white random noise images have been generated to stress how the behavior of algorithms varies with the percentage of foreground pixels (*density*) and minimum size of foreground blocks (*granularity*) [102]. Resolution is 2048×2048 for two-dimensional images, and 256×256×256 for three-dimensional volumes. Images and volumes were generated with the Mersenne Twister MT19937 random number generator, implemented in the *C++* standard [108]. 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, both for 2D and 3D.

Experiments available for GPU and 3D algorithms are the same described for CPU-2D. GPU test are based on the assumption that the input image is already in the GPU memory before the beginning of the algorithm, and that the output is required in the device memory as well. Therefore, the allocation of the output GPU image is considered in the total elapsed time, alongside potential allocation and deallocation of additional data structures that algorithms may need. Conversely, neither the allocation of the input image nor possible data transfers between host and device are considered. This also applies to every experimental result reported in this manuscript, unless otherwise specified.

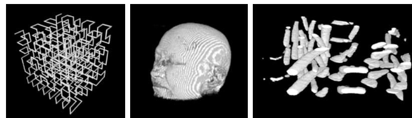


Figure 4.6: Samples of the YACCLAB 3D datasets. From left to right: Hilbert space-filling curve, OASIS and Mitochondria medical imaging data.

4.7 Conclusion

In this Chapter we described two contributions to the image processing community: a comprehensive dataset for comparing connected components labeling algorithms and a portable open-source *C++* project to test different algorithms on top of it. No new algorithms were proposed, but this tool allows any new improvement to be evaluated uniformly with respect to existing proposals.

We presented an analysis of some results to showcase the possibility of this project, and doing so we demonstrated that in many cases it is hard to find an algorithm which clearly dominates the others. Moreover, giving normalized figures such as *clock per pixel* is poorly significant, because it implies that that number does not depend on the machine. When changing the compiler changes the algorithm behavior, the number becomes clearly useless.

This is not to say that no comparison is possible. If an algorithm is always faster than another in all tried configurations, the conclusion is then obvious. The *reproducible research* movement is strongly affecting the Image Processing community, making scientific advances readily available to all researchers and practitioners. We strongly support this view and our effort is exactly aiming at letting everybody pick the best CCL algorithm for his needs.

The YACCLAB framework will be used for the evaluation of all the algorithms that will be introduced in the next Chapters.

Chapter 5

A New Paradigm for Sequential CCL Algorithms

In this Chapter three novel CCL algorithms are described. The first two —Optimized CCL with State Predicion (PRED in short) and Connected Components Labeling on DRAGs (DRAG in short)— introduce novel breakthrough ideas which are then extended and completed with the third algorithm: Spaghetti Labeling. Even though the improvement provided by PRED and DRAG is marginal, we included them in this work of thesis since they represent the basis of Spaghetti Labeling which significantly improves the state-of-the-art. The novel paradigm defined by these algorithms sets a cornerstone in the research field.

5.1 Optimized State Prediction

The Optimized State Prediction (PRED) is a novel paradigm for connected components labeling. It employs a general approach to minimize the number of memory accesses, by exploiting the information provided by already seen pixels, removing the need to check them again.

As shown in literature, the decision table which rules the scan step can

be conveniently converted to an optimal binary decision tree [52], in which internal nodes represent conditions on mask's pixels, and leaves represents actions to be performed on the current pixel of the provisional image (x in Fig. 2.1a), such as the creation of a new label, the assignment of an existing label to the pixel, or the merge of two existing labels. Usually, the same decision tree is traversed for each pixel of the input image, without exploiting values seen in the previous iteration, which, if considered, would result in a simplification of the decision tree for the pixel.

To go beyond this limitation, we compute a reduced decision tree for each possible set of known pixels; these reduced decision trees are then connected into a single graph, which rules the execution of the CCL algorithm on the whole image. This graph contains a starting tree, which should be accessed to process the first pixel of every row, and other trees, which are accessed to process the following pixels. Each leaf of a tree, which represent the action to be performed on a pixel, is connected to the root of a second tree which should be executed for the next pixel. The obtained graph can then be directly converted into running code.

5.1.1 Background

The algorithm described in [53, 54] is basically equivalent to the one in [98]. It uses a pixel based scanning with online equivalence resolution by means of a Union-Find technique with Path Compression (UFPC), plus a decision tree for accessing only the minimum number of already scanned labeled pixels.

In [52] it was proved that different versions of the decision tree are equivalent to the previous one and that, when performing 8-connectivity labeling, the scanning process can be extended to Block Based scanning, that is scanning the image in 2×2 blocks (BBDT). Building the decision tree for that case is much harder, because of the large number of possible combinations. In [60] a proved optimal strategy to build the decision tree has been proposed, by means of a dynamic programming approach. The final decision tree is generated automatically by another program.

Another variation of Block Based analysis was proposed in [106], which is reported to be faster than the previous one.

He *et al.* [48] recently observed that BBDT still checks many pixels repeatedly, because after labeling one pixel, the mask moves to the next one,

but many pixels in the current mask are overlapped to the previous ones, which may have already been checked.

They thus propose a Configuration-Transition-Based (CTB) algorithm which uses a set of different configurations to represent the current *state* of the algorithm and employ it to make further decisions. This allows to save a number of accesses to pixels and thus to speed-up the labeling process. The algorithm is specifically designed for the task and no provision to a general methodology is foreseen in the paper.

In the following of this Section we will show that our variation PRED is often faster than the algorithm from He [48], and even faster than [52] when tested on very low density images.

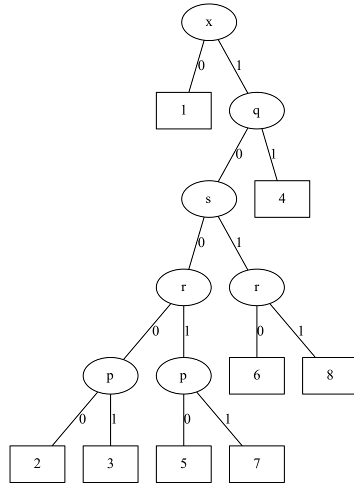


Figure 5.1: One of the possible full decision trees from the mask of Fig. 2.1a.

5.1.2 Method

We focus our analysis on 8-connectivity and start by observing the neighborhood mask of Fig. 2.1a, choosing one of the possible optimal decision trees we can obtain from it. Fig. 5.1 provides a visual representation of such a tree: the first thing to do is to check whether current pixel x is background or foreground (0 or 1). If $x = 0$, than we do not need to do anything (action 1) and simply move to the next pixel, otherwise we start looking its neighborhood. We start from q because it is connected to all other pixels: if $q = 1$ its label will be equivalent to that of all others, so we simply *assign* its label to x (action 4). If instead $q = 0$ we are in the case of potential *merge* of different previously unconnected components through x. To this aim, we need to check pixels p, s and r basically in any order we like with the only saving of avoiding checking p if $s = 1$ or viceversa,

since p is always connected to s. Finally, if no pixel is foreground, *i.e.* $p = r = s = 0$, we create a *new label* (action 2). In Fig. 5.1 actions 3, 5, and 6 are *assign* p, r, and s respectively, while actions 7 and 8 are *merge* $p + r$ and $s + r$ respectively.

Following [48], we observe that pixels pixels x will be the next s, q will be the next p, pixels r will be the next q. So if during the tree traversal we made a choice on x, q or r we know the values of s, p and q at the next step. This means that the corresponding subtree may be substituted to the choice made on that pixel.

Let's start with the first thing we know for sure: at the beginning of a line pixels p and s are 0 (there is no foreground outside the image). So all choices do not need to check those, meaning that we can remove the right branch of s and use its left subtree instead. There $p = 0$, thus we remove that check and substitute it with actions 2 and 5. This gives reduced tree A of Fig. 5.2. Its meaning is very clear to understand: first check pixel x: if it is foreground check the previous line (q and r), otherwise move next.

If we had a background pixel x before, we know that $s = 0$ so, again, we can remove its right branch thus obtaining the reduced tree B. If instead x was foreground and q was too, both $p = s = 1$ and we can remove the left branch of s getting reduced tree C. We keep going in this way and in many cases we obtain the same information from every leaf. Overall just two more different reduced trees are obtained (D and E in Fig. 5.2). It is noteworthy that, if in any tree we observed that $r = 1$, we move to tree D, which is simply a check on x: if it is foreground we assign the label of q without even the need of checking if it is foreground.

At every leaf of every tree, we know which tree shall be used next, so we mark that edge with a dotted line, to stress the difference with respect to the choices within the trees: inside every tree we just check the pixels, after performing the action in a leaf we need to advance the mask, check if the image row is finished and then proceed or break out of the current line.

Implementation

Moving from theory to practice is just a matter of translating every branch with a conditional statement and every dotted link with an unconditional jump.

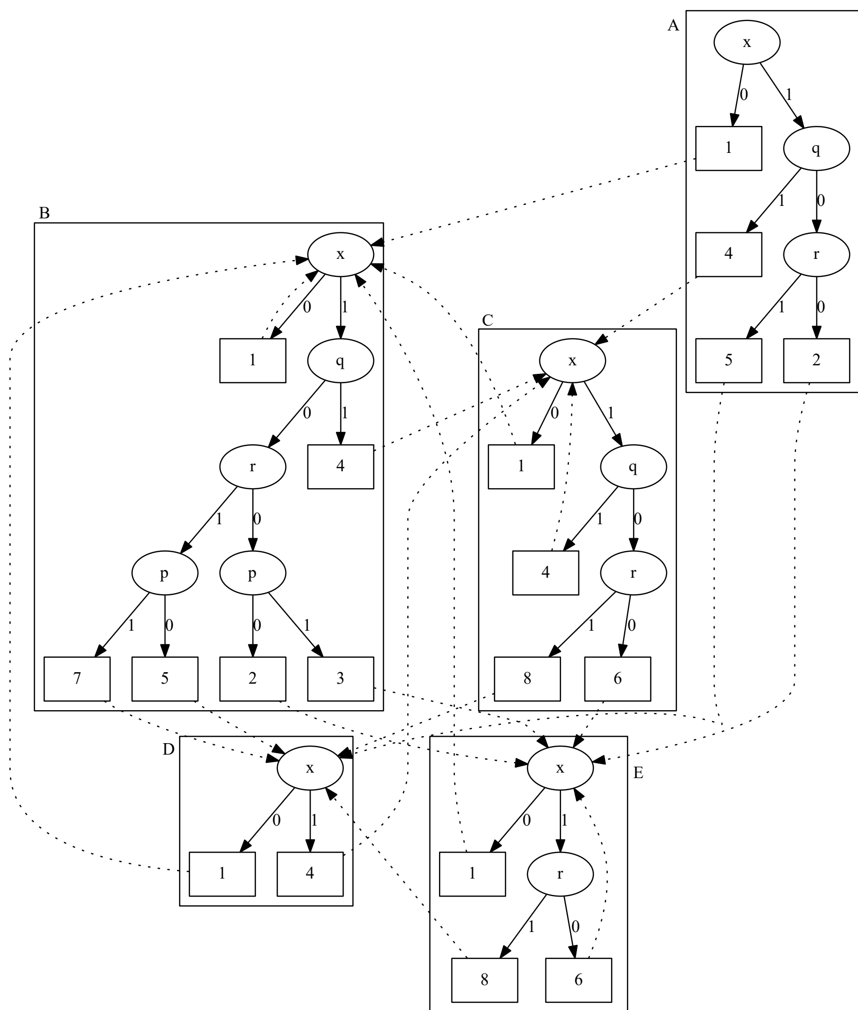


Figure 5.2: The final graph of decision trees, obtained from the full decision tree of Fig. 5.1.

At the beginning of every tree an increment of the current pixel position has to be made along with a check for the end of the row. A sample of the code in *C* language for tree C is provided in Listing 5.1. The real implementation of the actions is omitted and substituted with comments. Conditions are indicated with a macro which shall be defined by the specific implementation.

One thing should be noted: the *end of row* check stops one pixel before the end of the row. In this way we can make specialized versions for all the trees for the last pixel case, where we know that $r = 0$. This simple trick allows us to save an end of line check at every use of pixel r, since we already know that we are at least one pixel inside the image.

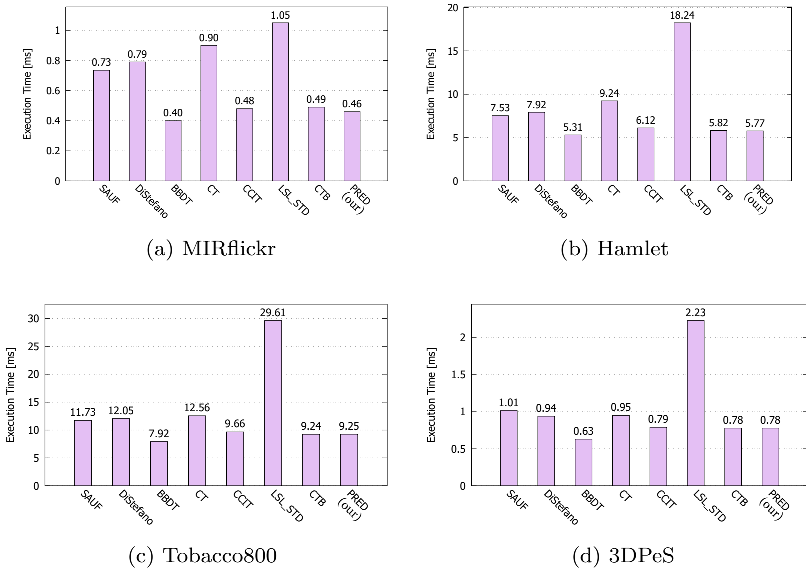
```
1  ...
2  tree_C:
3    if (++c >= w - 1)
4      goto break_C;
5    if (condition_x) {
6      if (condition_q) {
7        // action 4 - x <= q
8        goto tree_C;
9      }
10     else {
11       if (condition_r) {
12         // action 8 - x <= r + s
13         goto tree_D;
14       }
15       else {
16         // action 6 - x <= s
17         goto tree_E;
18       }
19     }
20   }
21   else {
22     // action 1 - do nothing
23     goto tree_B;
24   }
25 ...
```

Listing 5.1: Example code for reduced tree C. Here comments are used to indicate where the actions should be performed.

5.1.3 Experimental Evaluation

Tests were performed on a Windows PC with an Intel Core i7-4790K CPU @ 4.00 GHz and Microsoft Visual Studio 2013. All algorithms reported in this Section were included in YACCLAB. Tests were repeated ten times: for each image only the minimum execution time was considered (in order to reduce/avoid the influence of other tasks on the final results). Charts and Tables report average times for every dataset or density/size considering only the minimum execution time on every image.

In the following, we use acronyms to refer to the available algorithms: CT is the Contour Tracing approach by Fu Chang *et al.* [56], CCIT is the algorithm by Wan-Yu Chang *et al.* [106], DiStefano is the algorithm in [120], BBDT is the Block Based with Decision Trees algorithm by Grana *et al.* [52], LSL_STD is the Light Speed Labeling algorithm by Lacassagne *et al.* [105], SAUF is the Scan Array-based Union-Find algorithm by Wu *et*



(a) MIRflickr (b) Hamlet

(c) Tobacco800 (d) 3DPeS

Figure 5.3: Average results on a i7-4790K CPU @ 4.00 GHz with Windows and Microsoft Visual Studio 2013. Lower is better.

al. [54], CTB is the Configuration-Transition-Based approach described in [48]. Our method is denoted as PRED.

Fig. 5.3 and Table 5.1 report mean run-times for each considered dataset: MIRflickr, Hamlet, Tobacco800 and 3DPeS (see Section 4.2 for more details).

For what concerns synthetic images, the following conclusions can be drawn:

- *Size*, Fig. 5.4b: highlights a linear dependency of execution time with respect to the number of pixels. This is true for all algorithms except Di Stefano's one, which shows, as expected, a worse performance when the number of pixels is high.

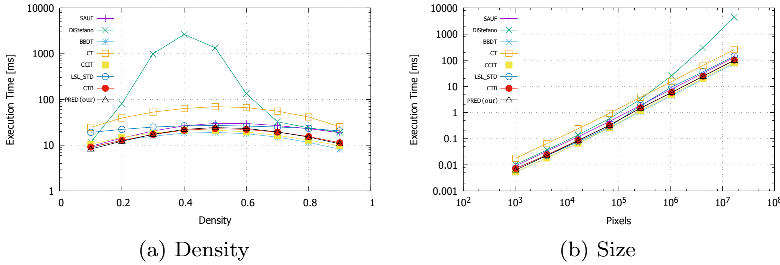
- *Density*, Fig. 5.4a: reports density tests performed on the syntethic dataset. Like almost all others algorithms, PRED shows an increased execution time on middle densities, because the number of labels and merges between equivalence classes is higher and also branch prediction can affect negatively the execution times. Di Stefano's algorithm produces the worst performance in the middle densities instead LSL_STD is the only one that demonstrated a quasi-linear trend, probably due to the lower number of conditional statements.

It is important to underline that all numerical values reported in the graphs and tables also consider times needed to define and initialize data structures, except for the binary input matrix. Indeed, if an algorithm

Table 5.1: Average results in ms on a i7-4790K CPU @ 4.00 GHz with Windows and Microsoft Visual Studio 2013. Lower is better.

	MIRflickr	*Hamlet*	*Tobacco800*	*3DPeS*
SAUF	0.735	7.531	11.728	1.014
DiStefano	0.795	7.921	12.047	0.945
BBDT	0.403	5.314	7.924	0.625
CT	0.902	9.245	12.561	0.953
CCIT	0.481	6.118	9.663	0.791
LSL_STD	1.052	18.242	29.608	2.234
CTB	0.491	5.819	9.237	0.778
PRED (our)	0.458	5.769	9.252	0.778

needs to save more information to compute correct labeling, these must be considered in the total amount of execution time.



(a) Density

(b) Size

Figure 5.4: Density (a) and size (b) tests on a i7-4790K CPU @ 4.00 GHz with Windows and Microsoft Visual Studio 2013. Lower is better.

5.1.4 Conclusion

In this Section we presented a novel approach for performing connected components labeling, which employs a reproducible strategy able to avoid repeatedly checking the same pixels multiple times. Experimental results are very promising and even if the current algorithm is not able to beat BBDT algorithm on the real datasets, it was faster on the synthetic one for low density cases. Moreover, it was able to surpass the performance of CTB which is currently the second best accordingly to the YACCLAB benchmark. In Section 5.3 we will apply a similar optimization strategy also to the BBDT algorithm. In that case the tree reduction cannot be performed *by hand*, given the enormous size of the decision tree. Moreover, some further details must be taken into account to achieve the final optimization goal.

The source code of the described method has been included in the YACCLAB benchmark, so that it will be possible to check the real performance on different machines and compare it with future proposals.

5.2 Connected Components Labeling on DRAGs

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

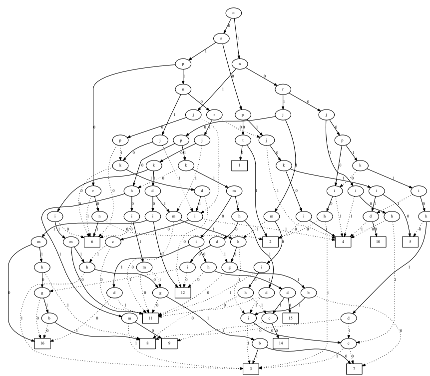


Figure 5.5: Example of optimal DRAG for the BBDT algorithm. Rectangles identify leaves (*i.e.* actions) and ellipses identify nodes (*i.e.* conditions to check). Continuous lines represent links belonging to the original DTree while dotted lines represent links generated during the transformation process which converts the DTree into a DRAG. The full size image is reported in Fig. 5.9.

In Very Large Scale Integration (VLSI) design, Binary Decision Diagrams (BDDs) have been used to model binary functions [126]. In order to have integer valued leaves, $f : \{0,1\}^n \rightarrow I$, Multi-Terminal BDDs (MTBDDs) have been introduced, where n is the number of decision variables (pixels in the mask) and I is the set of possible actions. In our case a further addition is needed, since the leaves need to have multiple alternative actions instead of just one. In this Section we introduce a novel approach to model decision problems as Directed Acyclic Graphs with a root, which we will call Directed Rooted Acyclic Graphs (DRAGs). In this structure we can model the decision outcome as a set of equivalent actions, as it was done for the DTree of BBDT algorithm [52]. The advantage of this different representation is that a DRAG will contain only the minimum number of nodes required to reach the leaf corresponding to a set of condition values. In a DTree the same set of conditions may be checked in multiple subtrees, while in a DRAG, being it a graph, these could be merged together. It is clear that this does not save any condition check, but the code generated from the DRAG will include the same checks only once, sensibly reducing

the number of machine instructions, thus the impact on instruction cache. In Section 5.2.3 we experimentally show the obtained improvement.

5.2.1 Modelling CCL with Decision Trees

							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 5.6: *OR*-decision table associated to the Rosenfeld mask (Fig. 2.1a).

In [52] the procedure of collecting labels and solving equivalences is described by a *command execution metaphor*: the current and neighboring pixels provide a binary command word (foreground is 1 and background is 0) which leads to the execution of a corresponding action. The possible actions are: “no action” if the current pixel is background, “new label” if it has no foreground neighbors, “assign” or “merge” based on the label of neighboring foreground pixels. The relation between the commands and the corresponding actions may be conveniently described by means of a *decision table* [127]. Additionally, in [52] an extension to classical decision tables is provided, *i.e.* every rule (binary word) is associated to a set of *alternative* actions. This is a peculiar characteristic of the CCL problem when multiple foreground neighbors share equivalent labels because these have already been merged in previous steps. This extension is called *OR*-decision table, opposed to the previous ones which required all actions to be performed (*e.g.* do action 1 *and* action 7 *and* ...). An example of *OR*-decision table, associated to the Ronsenfeld mask for the CCL task (Fig. 2.1a), is reported in Fig. 5.6.

An *AND*-decision table can be transformed into an optimal DTree by the use of the dynamic programming approach described in [128] by Schumacher *et al.* This process guarantees to obtain a DTree with the minimum average number of conditions to check in order to choose the correct action to be performed. Moreover, in [60] Grana *et al.* proved an optimal strategy

to extend the Schumacher algorithm to *OR*-decision tables, thus allowing to convert them into DTrees and from them into running code. If any leaf still had equivalent actions, a random one could have been picked. This automatic procedure empowers the possibility to extend the algorithm to more complex masks, such as the one reported in Fig. 2.1b. This mask has the advantage to allow the labeling of four pixels (o, p, s and t) at the same time, roughly reducing the cost of the *first scan* by a factor of four, and to reduce the number of merge operations since labels equivalence is implicitly solved within 2×2 blocks.

Fig. 5.7 reports the Optimal Decision Tree (ODT) obtained with the aforementioned procedure. The total number of nodes (ellipses) is 136 and leaves (rectangles) are 137. Differently from what previously presented in the literature [60], this tree still shows all the equivalent actions in its leaves. As already said, these could be chosen in any way when generating code, but we will exploit equivalences for further optimizations.

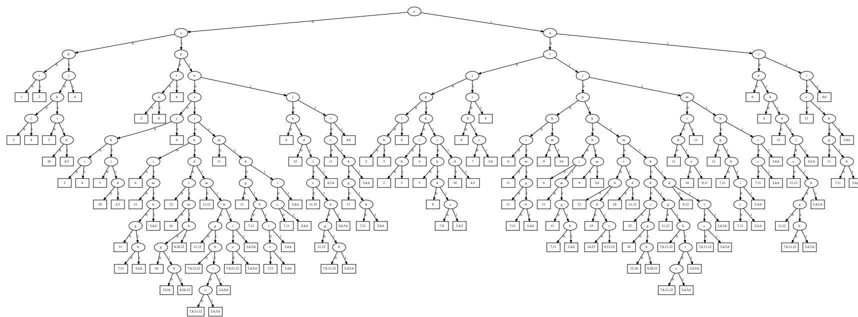


Figure 5.7: Optimal decision tree obtained with the mask of Fig. 2.1b. Best viewed online.

5.2.2 From Decision Trees to DRAGs

When looking at the assembly generated by the compiler from the *C* source code obtained from the ODT, we observed jumps which did not correspond to *goto*s in the source code. These were generated by the compiler optimizer in order to avoid repeating pieces of code which would have been identical. In fact, by looking at the tree, it is easy to spot *identical* subtrees which do

not require repetitions: the compiler is practically converting a tree into a Directed Rooted Acyclic Graph (DRAG).

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

Let us give a formal statement of the problem. We will call $\mathcal{DT}(C, A)$ the set of decision trees for the set of conditions C and actions A. These are full binary trees, *i.e.* rooted trees in which a vertex will either be a node with two children or a leaf without children. $\mathcal{N}$ is the set of nodes and $\mathcal{L}$ is the set of leaves. The condition of a node is denoted with $c(n) \in C$, with $n \in \mathcal{N}$, and the set of equivalent actions of a leaf is denoted with $a(l) \in \mathcal{P}(A) \setminus \{\varnothing\}$, with $l \in \mathcal{L}$. Each node n has a left subtree $\ell(n)$ and a right subtree $\reflectbox{$\ell$}(n)$, each rooted in the corresponding child of n.

Definition 5.2.2.1 (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 $\reflectbox{$\ell$}(r_1)$ is *equal* to $\reflectbox{$\ell$}(r_2)$.

Definition 5.2.2.2 (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 $\reflectbox{$\ell$}(r_1)$ is *equivalent* to $\reflectbox{$\ell$}(r_2)$.

A first transformation from a DTree to a DRAG can be performed by substituting all *equal* subtrees with a single instance, making every parent node point to that unique exemplar. Since $\mathcal{DT}$ equality is a transitive relation, we can traverse the tree and for every subtree search an equal one and immediately perform the substitution. The nice property of this transformation is that it does not depend on the order in which the original tree is traversed. The result of applying this *equal subtrees transformation* to the ODT of Fig. 5.7 is shown in Fig. 5.8. This DRAG has 86 nodes and

can be automatically converted to code by generating the subtree code only for “continuous” arcs and using *goto*s for “dotted” ones.

An even better transformation (*i.e.* with less nodes) is obtained by performing the same procedure, substituting equality with equivalence, and taking the intersection of actions in the corresponding leaves. Since all equal subtrees are also equivalent, all previous substitutions will be performed, plus, possibly, some more. Unfortunately, $\mathcal{DT}$ equivalence is

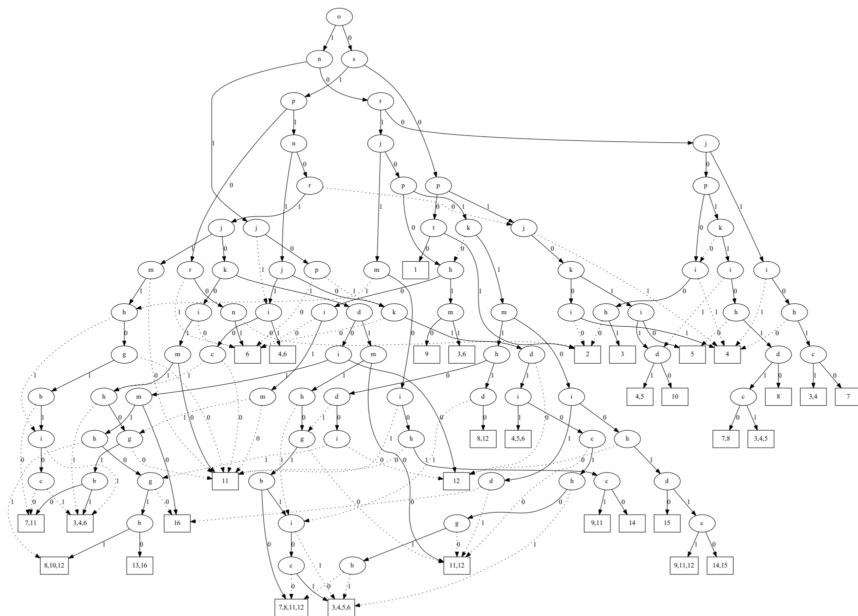


Figure 5.8: DRAG obtained using the equal subtrees transformation. Best viewed online.

not transitive, and it therefore depends on the order of traversal. Taking an intersection earlier may hinder the possibility of doing a better choice later, thus of minimizing the number of nodes. Of course, trying all possible traversal orders is unfeasible.

We already noticed that equal subtrees need to be substituted also when using the equivalence transformation, so instead of starting from the original tree, it is reasonable to start from the DRAG obtained by applying

the equal subtrees transformation. From this DRAG we can obtain many variations by selecting a single action from leaves with more than one, in all possible ways. In our particular case (Fig. 5.8), 11 leaves have two choices, 5 have three and only 2 have four. This gives a total of 7 962 624 different DRAGs. Since the number is not absurd, it is possible to reduce all the DRAGs with the equal subtrees transformation, and keep the one with the minimum number of nodes. This is also an optimal solution of the original problem. The result of applying such a reduction to the DRAG of Fig. 5.8 is shown in Fig. 5.9. The final DRAG has 72 nodes and 15 leaves with no intersection.

Figure 5.9: Example of optimal DRAG for the BBDT algorithm. Best viewed online.

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

5.2.3 Experimental Results

In order to evaluate the benefit of the proposed strategy we tested the DRAG algorithm using YACCLAB. Tests revealed that the impact of the labels solver on the overall performance is strictly limited for fastest algorithms, for this reason we will only report results obtained with the UFPC solver [53].

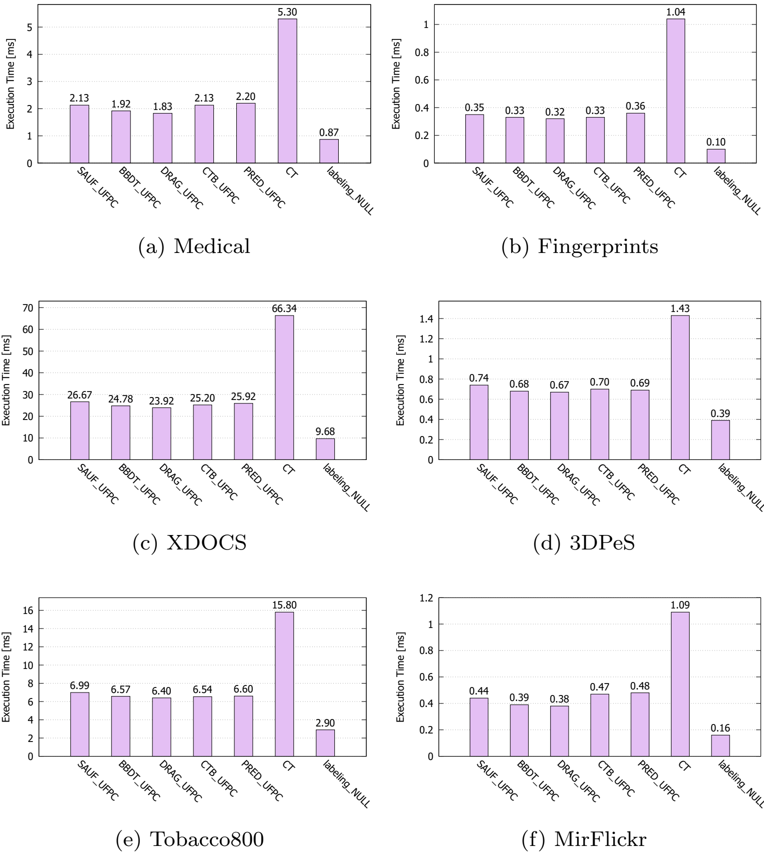
In the following, we will use acronyms to refer to the compared algorithms: CT is the Contour Tracing approach by Fu Chang *et al.* [56], SAUF is the Scan Array-based Union-Find algorithm by Wu *et al.* [54], BBDT is the Block Based with Decision Trees algorithm by Grana *et al.* [52], CTB is the Configuration-Transition-Based algorithm by He *et al.* [48], and PRED is the Optimized Pixel Prediction by Grana *et al.* [121]. Moreover, NULL is the lower bound limit defined in 4.3.1.

Fig. 5.10 compares the average execution time of the aforementioned algorithms on six different YACCLAB datasets [129]: Medical, Fingerprints, XDOCS, 3DPeS, Tobacco800, and MIRFlickr. The test was run on an Intel Core i7-4770 CPU @ 3.40GHz (4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache) with Linux OS and GCC 7.2.0 compiler enabling O3 flag. The behavior is practically equal on all datasets, with DRAG being always the winner. The second best is nearly always the version using BBDT, with the ODT previously shown. The lower impact on the instruction cache is beneficial, even in this case, in which the amount of available memory is much larger than required.

5.2.4 Conclusion

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

The missing step for this work is the ability of including the prediction stage introduced in the previous Sections within a unique framework, and being able of solving the complete problem in feasible time. This will be discussed in Section 5.3.



(a) Medical (b) Fingerprints

(c) XDOCS (d) 3DPeS

(e) Tobacco800 (f) MirFlickr

Figure 5.10: Experimental results obtained on an Intel Core i7-4770 CPU @ 3.40GHz, running Linux with GCC 7.2.0 with YACCLAB. Lower is better.

5.3 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, as shown in the previous Sections, 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, we combine a block-based mask with state prediction and code compression: the resulting algorithm is modeled as a Directed Rooted Acyclic Graph with multiple entry points, which is automatically generated without manual intervention. When tested on synthetic and real datasets, in comparison with optimized implementations of state-of-the-art algorithms, the proposed approach shows superior performance, surpassing the results obtained by all compared approaches in all settings.

5.3.1 Background

For what concerns CCL performance optimization, the first significant improvement has been provided by Wu *et al.* [53], 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. 2.1a). 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.* [52] subsequently introduced a major breakthrough, consisting in a 2×2 block-based approach (Fig. 2.1b), 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 [49], and few of them introduced significantly novel ideas, in particular:

– a proved algorithm to produce optimal decision trees [60];

- 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 [48];
- 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 5.1);
- 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 5.2).

Prediction, as introduced by He *et al.* [130] and later improved in [48] and [58], 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 [48] 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 [121] and described in Section 5.1 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 [58], an extension of [48]. For this reason, all works on prediction chose to avoid the complexity of the BBDT mask, and simplified it in various ways.

In this Section, 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 [129]. 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*.

5.3.2 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.

Optimal Decision Trees

In Section 5.2.1, we already explained how the procedure of collecting labels and solving equivalences can be described 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/b-

lock is background, *new label* if it has no foreground neighbors, *assign* or *merge* based on the label of neighboring foreground pixels/blocks. In [127], Schutte showed that *decision tables* may conveniently be used to describe the relation between commands and corresponding actions. Additionally, Grana *et al.* [52] 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. 2.1a), the associated *AND*/*OR*-decision tables are respectively reported in Fig. 5.11a and

							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

(a) AND

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

(b) OR

Figure 5.11: (a) *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. (b) *OR*-decision table for Rosenfeld mask.

Fig. 5.11b.

The *AND*-decision table can be converted into an Optimal Decision Tree (ODT) through the use of the dynamic programming approach introduced by Schumacher *et al.* [128]. 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 [60], 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. 2.1b. This mask allows the labeling of four pixels (o, p, s and t) at the same time, thus reducing the number of load/store and merge operations required. Indeed, labels equivalence is automatically solved within 2×2 blocks without requiring additional merges.

The ODT obtained with the aforementioned strategy and using Grana mask is reported in Fig. 5.7. Here, the total number of nodes is 136 and leaves are 137. Differently from what previously presented in [60], this tree still shows all the equivalent actions in its leaves, which will be exploited later for further optimizations.

State Prediction

As described in Section 5.1, He *et al.* [130] 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

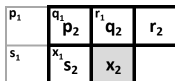


Figure 5.12: Unitary horizontal shift for Rosenfeld mask during image scan. Pixels named with “1” were inside the mask in the previous iteration while pixels named with “2” are currently inside the mask: $q_1 \rightarrow p_2$, $r_1 \rightarrow q_2$, and $x_1 \rightarrow s_2$.

be the next s, q will be the next p, and pixel r will be the next q. See Fig. 5.12 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.* [48] 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.

We, instead, proposed a general paradigm to leverage already seen pixels, which combines configuration transitions with the decision trees (Section 5.1). Algorithms that make use of a DTree usually traverse the same tree for each pixel of the input image. We noticed that the exploitation of values seen in the previous iteration could result in a simplification of the decision tree for the current pixel [121]. 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, we 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 algorithm, *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.

From Trees to DRAGs

In [46], we noticed the existence of identical and equivalent subtrees in the optimal decision tree obtained using the algorithm by [60]. We observed that identical subtrees were merged together by the compiler optimizer,

with the introduction of jumps in machine code (Section 5.2.2). 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. Then, we enhanced this optimization by merging not only identical subtrees, but also equivalent ones. The formal statement of the problem is found in Section 5.2.2.

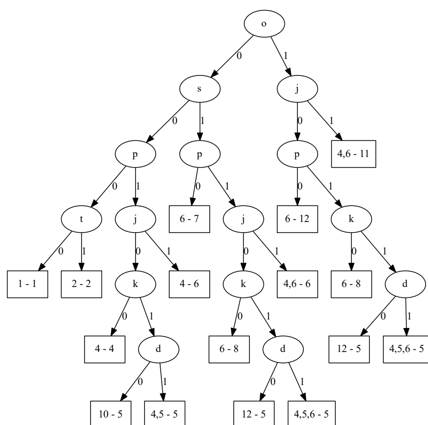


Figure 5.13: Reduced tree obtained from the optimal DTree (Fig. 5.7) through the application of the set of constraints $\{(h,0),(n,1),(i,1)\}$. Leaves contain equivalent actions to perform (to the left) and the index of the next tree to use (to the right) separated by a dash.

5.3.3 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.

State Prediction with Grana Mask

In the previous work by Grana *et al.* [121], 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. 5.12, 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. 5.13.

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. 5.14. 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.

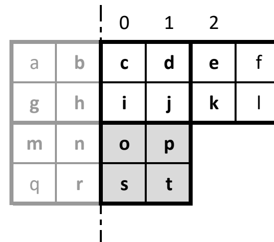


Figure 5.14: Grana mask configuration at the begin of a line. Dotted line represents the left border of the image.

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

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

The second problem is tied to the specific mask chosen by the CCL algorithm. When using BBDT mask, we notice that end of row constraints depend on whether the number of columns of the input image is even or odd (Fig. 5.15).

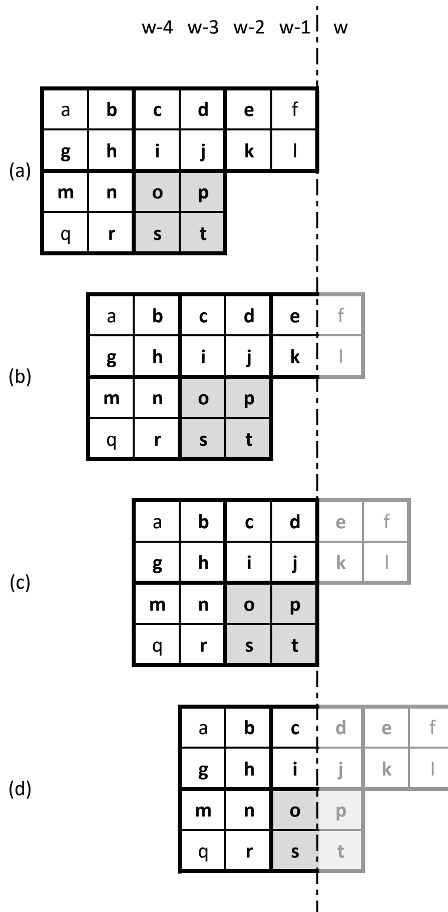


Figure 5.15: Possible configurations of Grana mask when it reaches the end of a line. The dotted line represents the right border of the image. Configurations (a) and (c) will occur when image has an even number of columns. Configurations (b) and (d) are required with odd number of columns.

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.

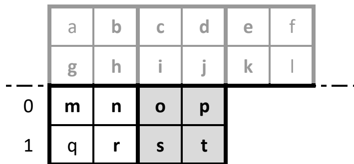


Figure 5.16: Grana mask configuration at the first line. The dotted line represents the upper border of the image.

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. 5.16). 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. 5.17.

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.* [46], 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.

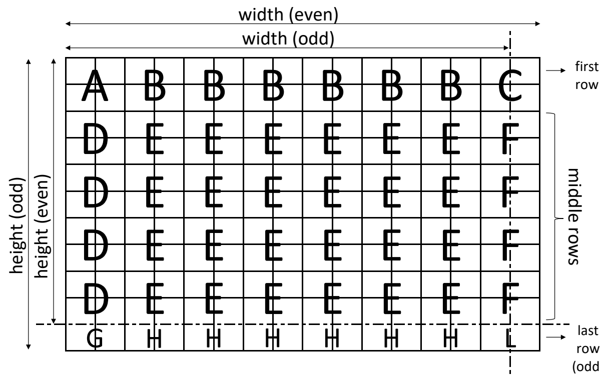
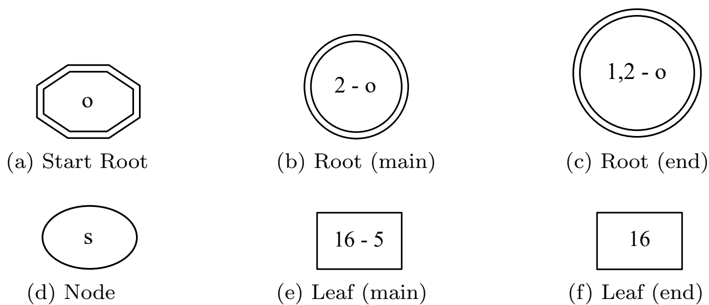


Figure 5.17: This figure shows which trees/forests should be used in each part of the image. For what concerns first row, "A" is the start tree, "B" represents the main forest and "C" is the end forest which automatically handle the odd/even number of cols. "D", "E", "F" have a similar meaning but for 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 5.3.2. 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 5.3.2, 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 in [46] 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 5.18: (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 5.3.3 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. 5.19 and Fig. 5.20.

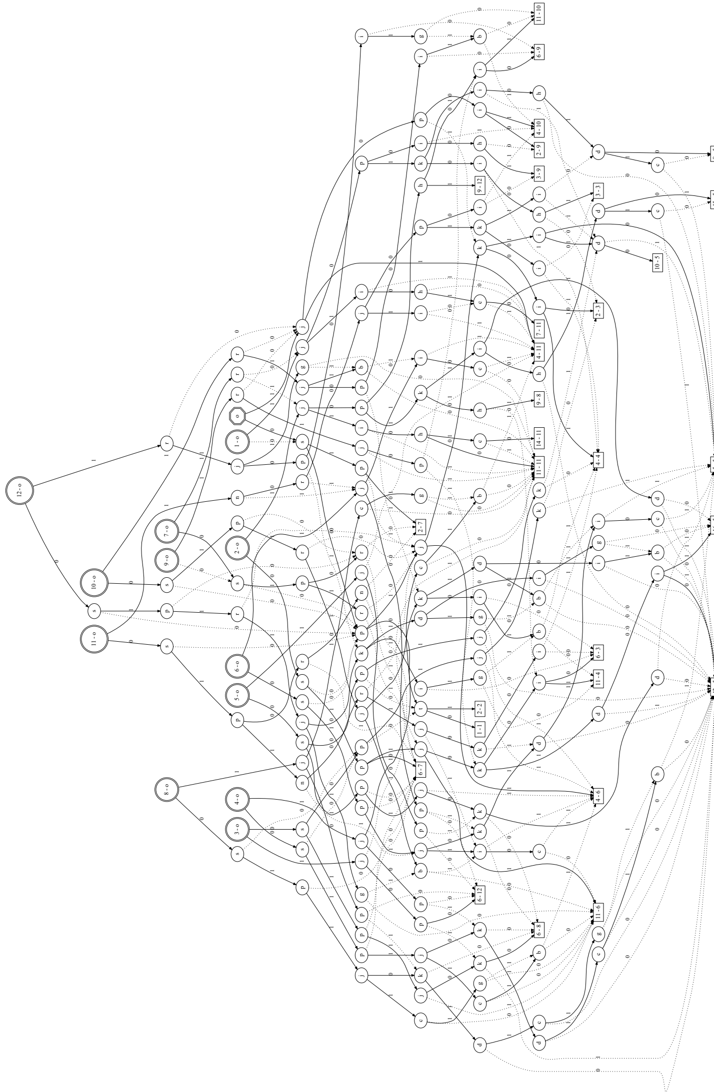


Figure 5.19: Forest of main trees connected into a single DAG with multiple entry points (roots). See Fig. 5.18 for details about notation used. Best viewed online.

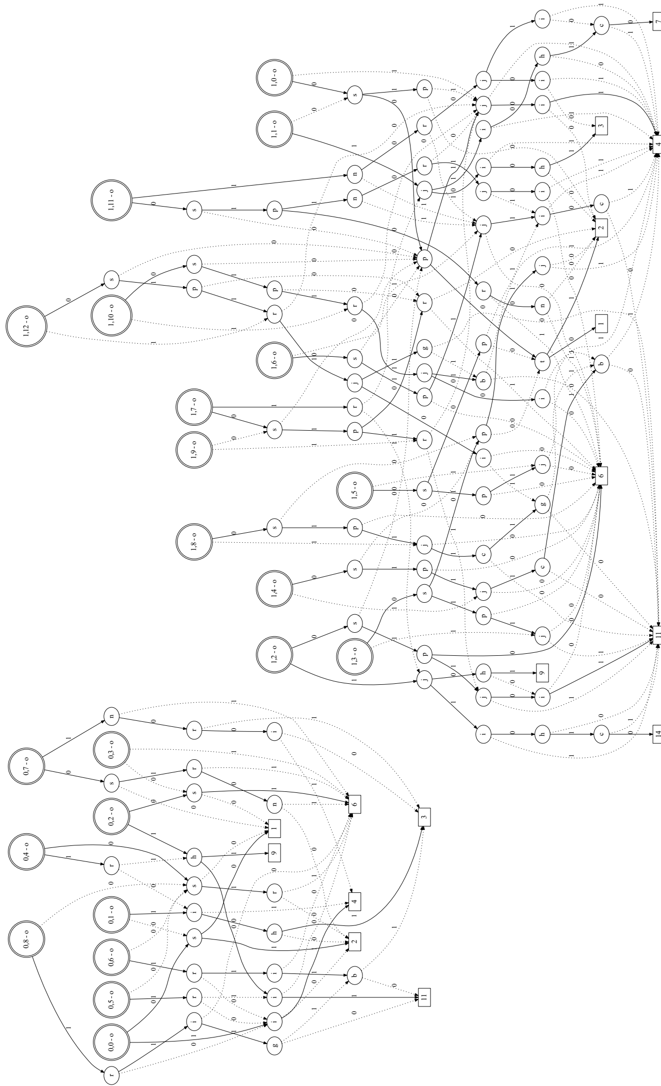


Figure 5.20: Forest of end trees connected into two groups of DAGs with multiple entry points (roots). See Fig. 5.18 for details about notation used. Best viewed online.

```
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_O) {
        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_O) {
            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_O) {
            if (CONDITION_N){
                    goto NODE_80; // Jump to existing subtree
            }
            else {
                    if (CONDITION_R){
...
```

Listing 5.2: *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.

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. 5.17.

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 5.2 exemplifies all the process.

5.3.4 Comparative Evaluation

In this Section the benefits of Spaghetti Labeling are evaluated using YACCLAB [129, 131, 132]. The fairness of the comparison is guaranteed by compiling the algorithms with the same optimizations and by running them on the same data and over the same machine.

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

The algorithms provided by YACCLAB cover most of the paradigms for CCL explored in the past. We selected the most significant ones, *i.e.* the best performing according to [129], in order to showcase the performance of the proposed algorithm.

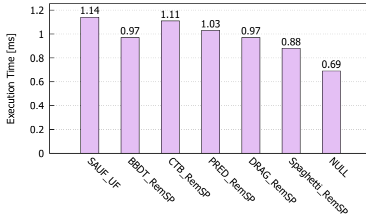
As usual, the following acronyms will be used to identify algorithms: SAUF is the Scan Array-based Union-Find algorithm by Wu *et al.* [54], BBDT is the Block-Based with Decision Trees algorithm by Grana *et al.* [52], CTB is the Configuration-Transition-Based algorithm by He *et al.* [48], PRED is the Optimized Pixel Prediction by Grana *et al.* [121], DRAG represents the Direct Rooted Acyclic Graph algorithm introduced by Bolelli *et al.* [46, 133] and described in Section 5.2. 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 [129]. Finally, the proposed method is identified as *Spaghetti*.

As stated in the previous Sections, 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 [129], 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) [99] for all the others.

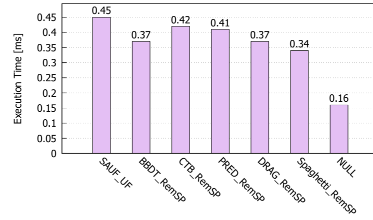
Average Run-Time Results

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 [129]. Experimental results are reported in Fig. 5.21.

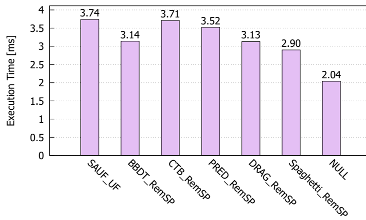
Algorithms that make use of Grana mask (BBDT, DRAG and Spaghetti) always perform better than those based on Rosenfeld mask (SAUF and PRED) or He mask (CTB), because of the lower number of merge operations and memory accesses on the output image and the equivalence data structures. Regarding the Rosenfeld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



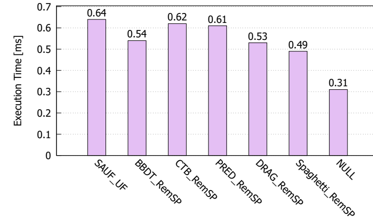
(a) 3DPeS



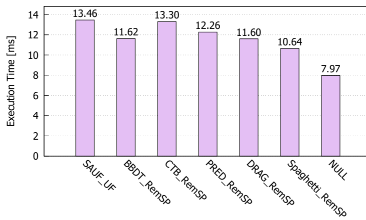
(b) Fingerprints



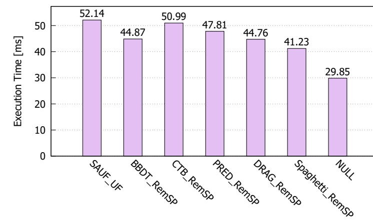
(c) Medical



(d) MIRflickr



(e) Tobacco800



(f) XDOCS

Figure 5.21: 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. 5.22. 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 [129], and this explains why execution times in Fig. 5.22 are higher than the ones in Fig. 5.21e. 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:

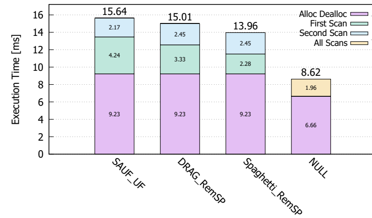


Figure 5.22: Average run-time tests with steps in ms on the *Tobacco800* dataset, obtained on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler. Lower is better.

– 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.

Load/store Operations

In Table 5.2, 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 5.2: 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.

Synthetic Images

Following a common practice in literature [48, 52, 121, 129], 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. 5.23. 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.

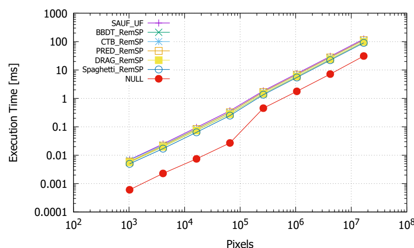
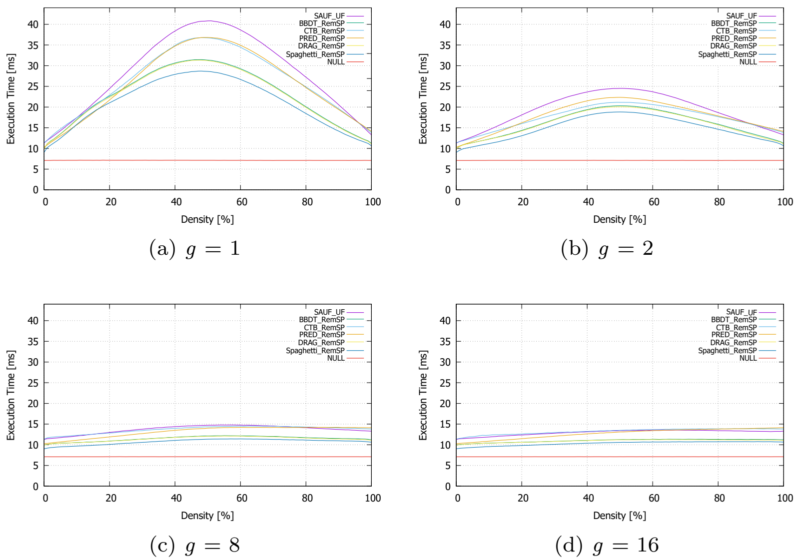


Figure 5.23: Experimental results in ms on an Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with Windows 10.0.17134 (64 bit) OS and MSVC 19.15.26730 compiler obtained using images of increasing size. Lower is better.

On the other hand, results obtained with different values of granularity (1, 2, 8, and 16) are depicted in Fig. 5.24: 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 5.24: 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.

5.3.5 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 and it is available in [131].

Chapter 6

CCL in Parallel Environments

This Chapter aims at providing four different contributions to the Image Processing community, describing novel strategies to label connected components in CPU- and GPU-based parallel environments. A strategy to parallelize existing sequential algorithms exploiting CPU multithreading is firstly presented in Section 6.1. Then, Section 6.2 aims to extend state-of-the-art GPU-based algorithms from 4- to 8-connectivity and to improve them with additional optimizations. In Section 6.3 the problem of labeling connected components on GPUs is addressed for the first time in literature using decision trees, revealing surprisingly good performance. Finally, Section 6.4 introduces two novel algorithms that outperform all the competitors, thus setting the state-of-the-art on GPU at the time of writing this thesis.

6.1 Multi-Cores Architectures

Modern computer architectures are multi-cores and support multi-threaded applications, so it is convenient to analyze how much the performance of CCL algorithms scale up when multi-threading is involved. The goal is to spread the computational cost among different threads without increasing the execution time with heavy synchronization mechanisms. A basic

approach to parallelize two scan labeling algorithms divides images into stripes (chunks) and operates with the following three steps [134]:

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

Assuming to have n workers available to compute labeling, we can spread the computational cost for the first scan between them dividing the image in n parts. In order to be cache friendly we choose to cut the image horizontally. Each stripe is then labeled independently (first scan) by each worker using provisional labels. To avoid getting overlapped labels among different stripes, each thread must use a different set of initial provisional labels. When using 8-connectivity, each 2×2 block of pixels can only contain a single label, so the starting label number for a given thread can be easily calculated as

$$\left\lfloor \frac{r_i + 1}{2} \right\rfloor \cdot \left\lfloor \frac{w + 1}{2} \right\rfloor + 1, \tag{6.1}$$

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

To avoid collisions each stripe deals with a disjoint set of labels and this implies non overlapped accesses to the array storing the equivalences.

(a) (b) (c)

Figure 6.1: (a) Example of binary image depicting text, (b) its labeling considering only the first scan of the parallel implementation of SAUF (two threads), and finally (c) its labeling result.

In Fig. 6.1b an example of the resulting image from the first scan is reported when two threads are involved.

After the first scan we have to establish a bridge between two adjacent chunks with the `Union` operation. This is required since chunks are labeled with uncorrelated provisional labels. In order to compute Merge we use masks reported in Fig. 6.2. There are basically two strategies to solve this problem and they are listed below.

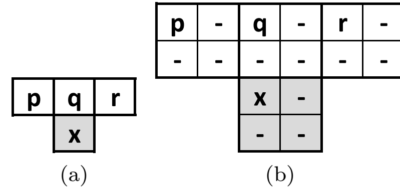


Figure 6.2: (a) Merge mask used by SAUF parallel algorithm and (b) by BBDT parallel algorithm.

- *Sequential Merge* scans each chunk's border sequentially and does not employ multi-threading, avoiding collision management in label equivalences array.
- *Logarithmic Merge* already proposed in [134], employs multi-threading to speed-up this operation. Indeed, we can operate concurrently on two or more chunks when they are not accessed at the same time by another thread.

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

The second approach, compared to the Sequential one, leads, at least in theory, to better performance. However, the management of Logarithmic Merge is more complex when the number of threads is not a power of two, so a correct implementation of it must include additional conditional statements. Hence, as shown in Table 6.1 the Sequential Merge operation performs better if compared with Logarithmic. Due to this, all parallel results shown in Section 6.1.1 are obtained using Sequential Merge.

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

To conclude the labeling process, the second scan is done in parallel by the same number of threads and using the same mask of the first scan.

Second scan better suits the multi-threading approach because it contains few conditional statements.

Table 6.1: Comparison between Sequential and Logarithmic Merge incidence (in percentage) with respect to the total execution time. The reported values are calculated considering all YACCLAB datasets, with different number of threads (hence chunks) and using BBDT parallel algorithm.

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

6.1.1 Experimental Evaluation

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

In the following of this Section, we use acronyms to refer to the available algorithms: BBDT is the Block Based with Decision Trees algorithm by Grana *et al.* [52], BBDT_$\langle n \rangle$ is the parallel implementation of the BBDT algorithm run with n threads, SAUF is the Scan Array-based Union-Find algorithm by Wu *et al.* [54], and, finally, SAUF_$\langle n \rangle$ is the SAUF parallel version run using n threads [1].

The parallel implementation of both BBDT and SAUF is based on the OpenCV's build-in $parallel_for_$ function. This function runs the parallel loop of one of the available parallel frameworks, selecting it at compilation time. All parallel results presented in this Chapter are obtained using *Intel Threading Building Blocks* (TBB).

Table 6.2 shows the average execution time of sequential and parallel version of both the algorithms on all dataset and with different number of threads. Experiments reveal that the overhead introduced by the

[1] The parallel implementation of BBDT and SAUF here described have been included in the OpenCV library after the publication of our paper [135]

*parallel_for_* with TBB is negligible *i.e.* the execution time of a sequential algorithm and its parallel version implemented with OpenCV parallel framework and tested with one thread is the same. As shown in Table 6.2, the speed-up obtained with two threads on SAUF is $\times 1.5$ in average and it increases up to $\times 4$ on random dataset when 12 threads are involved. Starting from 16 threads (*i.e.* when hyper-threading is involved and threads share cache of both first and second level) the performance of SAUF decrease. This could be explained by the increment of instruction cache misses due to data overflow. For what concerns BBDT algorithm, experimental results demonstrate a greater speed-up with low number of threads (*i.e.* $\times 1.7$ with 2 threads, up to $\times 4.7$ on random dataset with 8 threads). However, increasing the parallelism, as it happens with the SAUF algorithm, leads to worse performance. Once again this behavior could be related to the increase of instruction cache misses that, in this case, occur starting from 8 threads, because BBDT code footprint is much bigger than SAUF one.

Table 6.2: Average results in ms on a virtual Windows workstation with two Intel Xeon X5650 *CPU* @ 2.67GHz (6 physical cores and 12 logical processors per socket) and Microsoft Visual Studio 2013. Lower is better.

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

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

6.1.2 Conclusion

In this Section we demonstrated that employing multi-threading it is possible to spread the computational cost among different processes and improve the performance of two well known CCL algorithms: SAUF and BBDT. The same strategy can be applied to all two scan algorithms, including those proposed in Chapter 5.

6.2 Optimizing GPU Algorithms

The development of parallel algorithms to solve common problems is of growing interest, lead by the fast development of parallel hardware architectures. A simple process to parallelize sequential algorithm on CPU, consists of dividing the input image into horizontal stripes and computing labeling separately on each of them, as described in the previous Section.

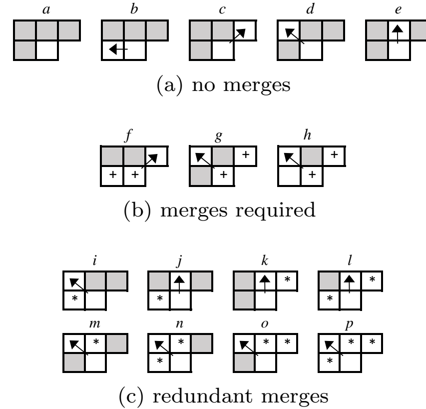


Figure 6.3: Possible neighbourhood configuration of a foreground pixel. Grey squares are background and white are foreground. Arrows show connections performed in *initialization* phase, crosses identify mergers that must be performed during *reduction* phase, stars highlight pixels to whom the merging operation can be delegated.

Computational throughput of Graphical Processing Units (GPUs) makes them eligible for many image processing methods that can be easily implemented on such architectures. The use of GPUs usually provides good performance, but this does not happen for less regular problems, such as CCL. Indeed, labeling GPU implementations tend to achieve comparable performance with respect to the CPU ones [67]. However, data transfer between device and host is very expensive, usually higher than the cost of labeling procedure. Therefore, all the applications that entirely execute on GPU would benefit from an optimized GPU-based labeling algorithm, removing the need for data transfers.

As introduced in Section 2.1.2, many approaches to compute GPU-CCL have been published in the last decade.

Most of them have focused on 4-connectivity only, even though many computer vision tasks require 8-connectivity CCL. According to the *Gestalt Theory* of perception, indeed, our senses operate the closure property per-

ceiving objects as a whole, even if they are loosely connected as happens in the 8-connectivity case [52].

This Section aims to improve state-of-the-art GPU algorithms with three main contributions: *(i)* extension from 4- to 8-connectivity of UF and KE methods *(ii)* optimization of KE 8-connected, and *(iii)* exhaustive evaluation of the proposed strategies and comparison with state-of-the-art using the YACCLAB benchmark.

6.2.1 Related Work

For better describing the algorithmic solution introduced in this Section, we need to detail some of the algorithms already introduced in Section 2.1 and Section 3.2.

Algorithm 3 *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
```

Union-Find Algorithm

Oliveira *et al.* proposed in [64] 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 [93]. 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* operations 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.

A possible implementation of `Find` and `Union` procedures is shown in Algorithm 3. Atomic operations are used in *union* to avoid problems concerning the simultaneous updates performed by different threads.

Algorithm 4 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`$(r, c)) + 1$

UF algorithm is based on the *union-find* data structure and it consists of three kernels: *initialization*, *merge* and *analysis*, described in Algorithm 4. 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.

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 [66], which employs a 4-connectivity, introduces additional improvements to the UF algorithm. This method consists of four steps: *initialization*, *analysis*, *label reduction*, and *analysis* again, which are detailed in the following:

(a) The *initialization* kernel, shown at line 1 of Algorithm 5, 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.

(b) 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.

Algorithm 5 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
```

(c) In *reduction* kernel, shown at line 11 of Algorithm 5, 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 foreground, only north has been chosen during *initialization* phase. The merging is performed by the `Reduce` procedure, which has the same effect of `Union`.

6.2.2 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. 6.3. Configurations in Fig. 6.3a 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. 6.3b 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. 6.3c 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 [67], 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 6. 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 6 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))
```

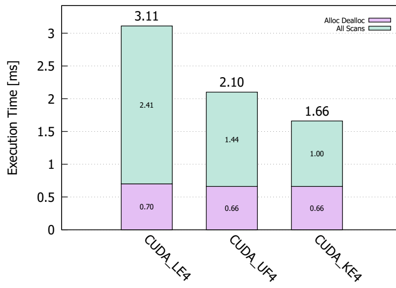
6.2.3 Experimental Results

In this Section, the performance of the proposed strategies are evaluated and compared with other state-of-the-art GPU-CCL algorithms. There are many variables that could influence the performance of an algorithm in terms of execution time: the machine architecture and the operative system on which test are performed, the adopted compiler, code implementation and last but not least the data on which algorithms are tested. In order to produce a fair comparison, our proposals have been evaluated with YACCLAB [129, 132]. We select the most significant real case scenarios datasets: *MIRflickr* [110], *Medical* [112], *Tobacco800* [113], *XDOCS* [114, 39], *Fingerprints* [116], and *3DPeS* [118]. A complete description of these datasets can be found in Section 4.2.

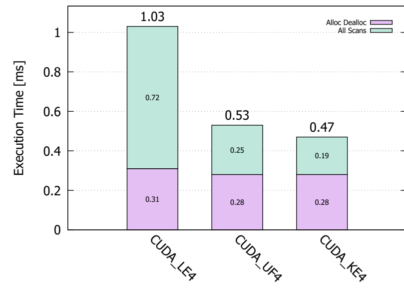
The algorithms have been implemented with Visual Studio 2017 using CUDA 9.2 and compiled for x64 architectures employing MSVC 19.13.26132 and NVCC V9.2.148 compilers with optimizations enabled. Tests are performed on an Intel Core i7-4770 CPU @ 3,40 GHz (with 4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache) with 16 GB of RAM available, and with a Quadro K2200 NVIDIA GPU with Maxwell architecture, 640 CUDA cores and 4 GB of memory. Both 4- and 8-connectivity implementations are provided for each algorithm except for BE, which employs a block-based strategy suitable for 8-connectivity only. A complete descrition of this algorithm is found in Section 6.4.4.

In Fig. 6.4 and Fig. 6.5 the average execution time for each algorithm and dataset is reported, respectively for 4- and 8-connectivity. Bar charts report separately the time needed for allocating data structures and the execution time required by the labeling procedure. According to [129], the allocation time reported in this charts is an upper bound of the real allocation time required by an algorithm on a given environment.

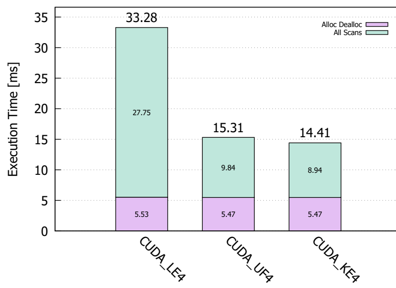
Focusing on Fig. 6.4, the allocation time is the same for each strategy, but for LE. Indeed, all strategies must allocate memory for the output image and LE requires an additional boolean flag to stop iteration when no change on the output image occurs. A similar conclusion can be drawn for Fig. 6.5, but the allocation time of BE is almost twice the others, since it relies on additional matrices to store equivalences between blocks and their labels. The execution time of the LE core phases is always the worst, given that it requires multiple iterations over the input image to update the output one until convergence.



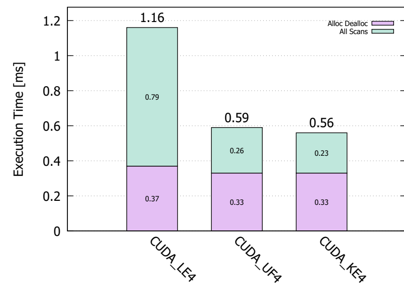
(a) Medical



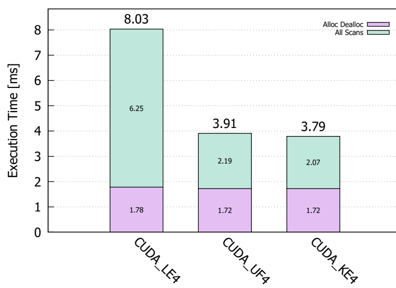
(b) Fingerprints



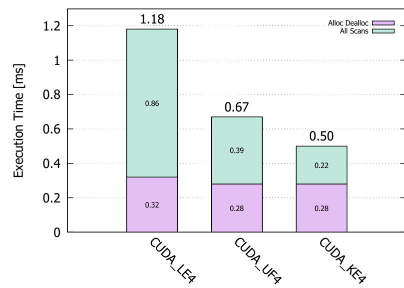
(c) XDOCS



(d) 3DPeS

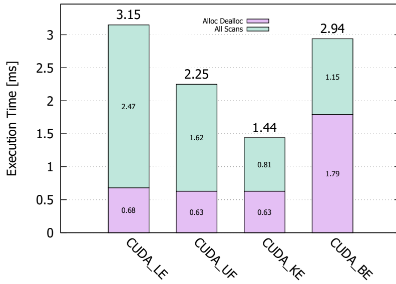


(e) Tobacco800

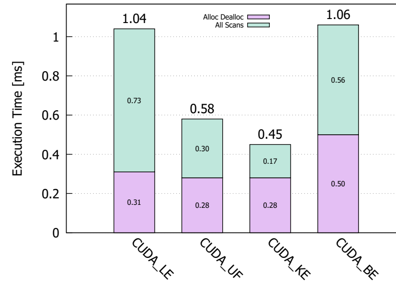


(f) MirFlickr

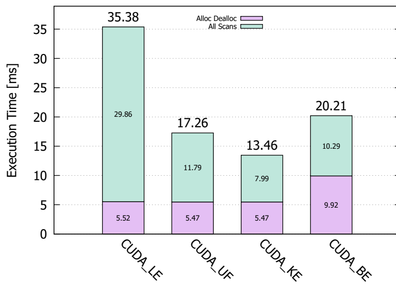
Figure 6.4: 4-connectivity experimental results obtained on a Nvidia Quadro K2200 GPU with the YACCLAB benchmark. For each dataset and algorithm the average execution time in ms is reported. Lower is better.



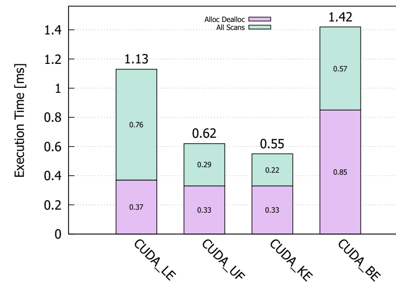
(a) Medical



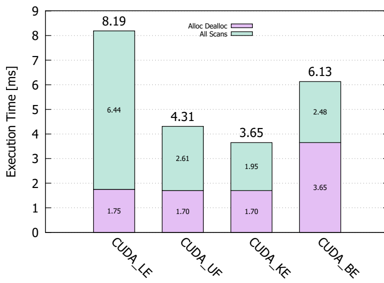
(b) Fingerprints



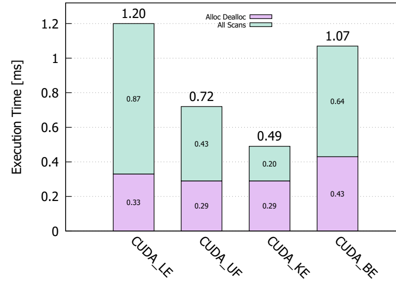
(c) XDOCS



(d) 3DPeS



(e) Tobacco800



(f) MirFlickr

Figure 6.5: 8-connectivity experimental results obtained on a Nvidia Quadro K2200 GPU with the YACCLAB benchmark. For each dataset and algorithm the average execution time in ms is reported. Lower is better.

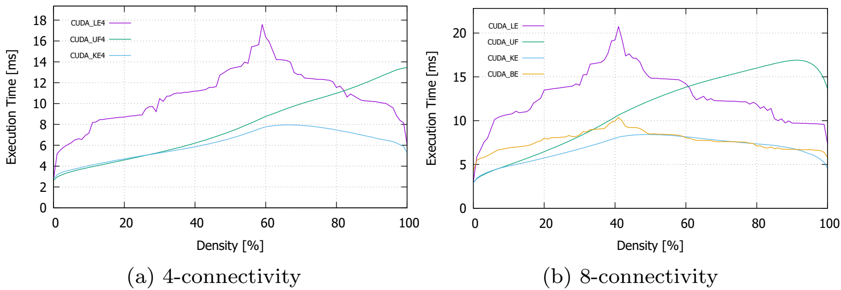


Figure 6.6: Execution time of (a) 4- and (b) 8-connectivity algorithms on images of increasing density.

The block scan approach introduced by BE allows to reduce by a factor of four the operations required by LE, thus reducing the labeling time at the expense of allocation step. In most cases, using blocks to scan images improve the performance, except for images with very low foreground density such as in 3DPeS.

Anyway, multiple scan approaches are always worse than others. As can be seen from charts in Fig. 6.5, the reduction in the number of iterations allowed by UF always improves performance. Moreover, KE allows the removal of the initialization phase of UF, always providing the best performance on real cases datasets.

In order to highlight strengths and weaknesses of the algorithms, and following a common approach in literature [52, 53, 57], an additional test has been performed on images with increasing foreground density (Fig. 6.6).

Considering both 4- and 8-connectivity, the LE approach has an increasing trend in the execution time up to 40% of foreground density, and a decreasing one after 60%. In the middle densities the execution time reaches the maximum value. This is linked to the number of iterations required by the labeling procedure to converge, as shown in Table 6.3. More specifically, the pixels patterns which cause the highest number of iterations appear near 60% of foreground density for 4-connectivity variation and near 40% for the 8-connectivity. The BE algorithm has a similar behavior, albeit with better performance. This confirms results on real cases datasets described before.

On the other hand, the execution time of the UF approach grows with

foreground density. This is because each pixel thread has to perform one *merge* operation for each connected pixel, and the number of those pixels is linked to density. The more merges there are, the more memory accesses and atomic operations are performed.

Performance gap between UF and KE significantly increases from 4- to 8-connectivity. This can be easily explained considering the optimization we introduced on the 8-connected variation of the algorithm. Indeed, thanks to `Reduction` procedure we are able to remove unnecessary operations, thus reducing overall execution time.

6.2.4 Conclusion

In this Section the problem of connected components labeling on GPUs is explored. Many approaches have been introduced over the years but we focused our study on four main methods: Label Equivalence, Block Equivalence, Union- Find and Komura Equivalence. All this algorithms have been reimplemented in their original 4-connectivity version, but BE that is suitable for 8-connected applications only. Moreover, we adapted both the Union-Find and Komura Equivalence strategies to the 8-connectivity, and introduced an optimization on the second one which reduces the memory accesses, improving the performance. Experimental results obtained with YACCLAB revealed the effectiveness of our proposals. The source code of described algorithms is available at [131], so anyone can download, test it on his own setup, and verify our claims.

Table 6.3: Average number of iterations required by LE implementing 8-connectivity on images of increasing density.

density (%)	0	10	20	30	40	50	60	70	80	90	100
iterations	1.0	4.0	5.0	5.0	7.2	5.0	5.0	4.0	3.9	3.0	2.0

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 5), 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 [136]. 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 [137].

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 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. Unfortunately, the *first scan* cannot

Algorithm 7 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 INITIALIZATION(L)
 2:     L[id_x] ← id_x

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

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

 7: kernel FINALLABELING(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
```

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 la-

bel assigned to each block into its foreground pixels. This job is performed in *FinalLabeling* kernel. Table 6.4 sums up the structure of the proposed algorithms, while Algorithm 7 provides a possible implementation of the described kernels.

Table 6.4: 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

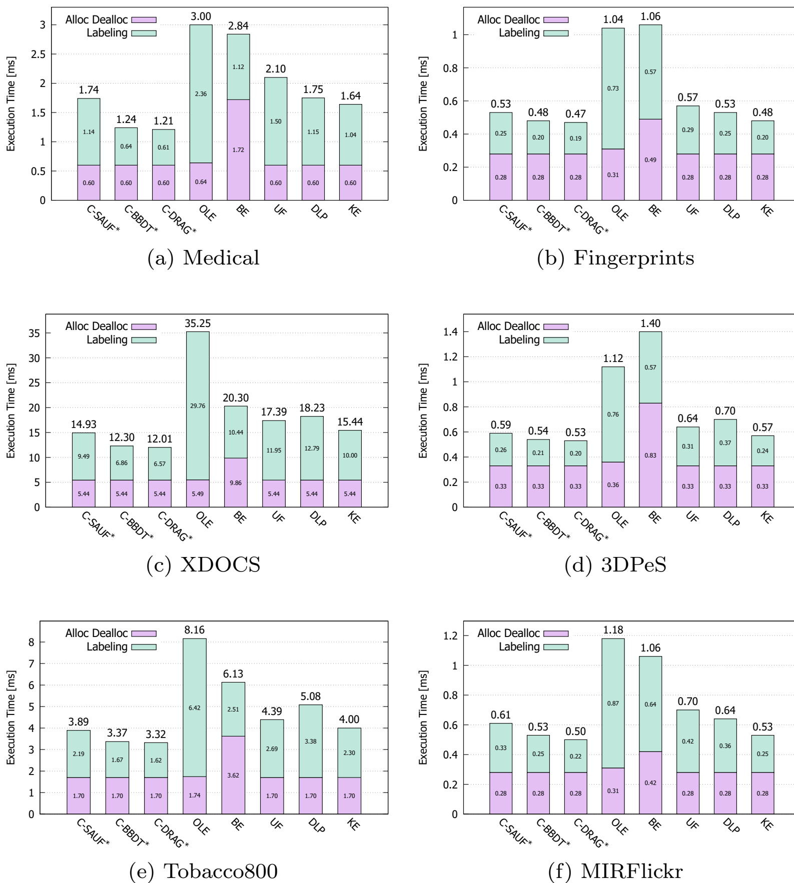
6.3.3 Comparative Analysis

In order to produce a fair comparison, algorithms are tested and compared with state-of-the-art GPU implementations using the YACCLAB open-source benchmarking framework [129, 132].

Experiments are performed on a Windows 10 desktop computer with an Intel Core i7-4770 (4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache), 16 GB of RAM, and a Quadro K2200 NVIDIA GPU (640 CUDA cores and 4 GB of memory). Algorithms have been compiled for x64 architectures using Visual Studio 2017 (MSVC 19.13.26730) and CUDA 10 (NVCC V10.0.130) with optimizations enabled.

With the purpose of stressing algorithms behaviours, thus highlighting their strengths and weaknesses, we perform three different kind of tests on many real case and synthetically generated datasets provided by YACCLAB.

As shown in Table 6.5, KE is confirmed to be the state-of-the-art GPU-specific algorithm, when taking into account the average run-time on all datasets. It is interesting to note that even a straightforward implementation of SAUF (C-SAUF) is able in many cases to outperform it.



(a) Medical (b) Fingerprints

(c) XDOCS (d) 3DPeS

(e) Tobacco800 (f) MIRFlickr

Figure 6.7: Average run-time results with steps in ms. Lower is better.

The two more complex tree-based algorithms (C-BBDT and C-DRAG) significantly reduce the computational requirements, with C-DRAG being always the best.

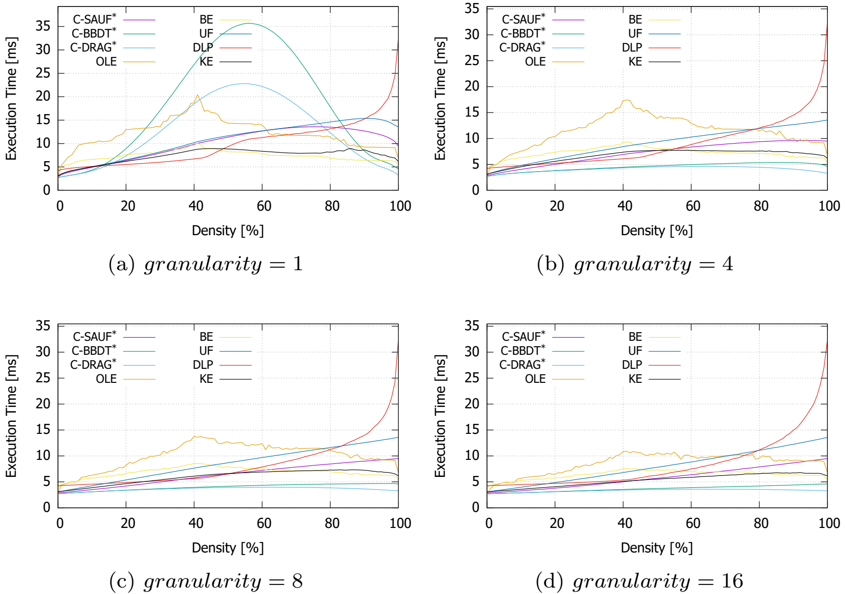
To better appreciate this results, it is useful to split the time required by memory allocation and the computation. Fig. 6.7 shows that in some cases more than 50% of the time is dedicated to memory allocation.

Table 6.5: Average run-time results in ms. The bold values represent the best performing CCL algorithm. Our proposals are identified with *. Lower is better.

	3DPeS	*Fingerprints*	*Hamlet*	*Medical*
C-SAUF*	0.560	0.487	2.867	1.699
C-BBDT*	0.535	0.472	2.444	1.249
C-DRAG*	**0.526**	**0.460**	**2.423**	**1.220**
OLE [62]	1.111	1.031	5.572	2.996
BE [63]	1.401	1.056	4.714	2.849
UF [64]	0.593	0.527	3.243	2.062
DLP [69]	0.657	0.484	3.323	1.719
KE [68]	0.565	0.478	2.893	1.644
	MIRflickr	*Tobacco800*	*XDOCS*	
C-SAUF*	0.571	3.846	14.870	
C-BBDT*	0.529	3.374	12.305	
C-DRAG*	**0.496**	**3.322**	**12.012**	
OLE [62]	1.174	8.152	35.245	
BE [63]	1.053	6.120	20.314	
UF [64]	0.656	4.332	17.333	
DLP [69]	0.597	5.031	18.182	
KE [68]	0.523	4.007	15.445	

BE clearly suffers from the additional data structures. When moving to larger images (*e.g.* XDOCS), the reduced time allowed by C-BBDT and C-DRAG is evident. How is this possible? Irregular patterns of execution are supposed to slow down the thread scheduling, so a decision tree is the worst thing that could happen to GPUs. This is not a myth, but we need to understand how often different branches are taken in the execution. So, the third test (Fig. 6.8) is run on a set of synthetic images generated by randomly setting a certain percentage (*density*) of pixel blocks to foreground: *granularity* test. When granularity is 1 (Fig. 6.8a), it is possible to observe that both C-BBDT and C-DRAG computational time explodes around density of 50%, in accordance with our expectation. The only cases in which those algorithms outperform GPU-specific ones is when density is below 10% or over 90%. Indeed, at low/high densities the decision is taken quickly 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 densities,

this explains the previously observed behavior. Moreover, when granularity grows (Fig. 6.8b-Fig. 6.8d), 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.



(a) $granularity = 1$

(b) $granularity = 4$

(c) $granularity = 8$

(d) $granularity = 16$

Figure 6.8: Granularity results in ms on images at various densities. Lower is better.

6.3.4 Conclusion

> It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.
>
> Sherlock Holmes

In this Section we have addressed the problem of connected components

labeling on GPUs from a different perspective. We have focused our analysis on three 8-connectivity sequential algorithms relying on decision trees and we have adapted them to GPU programming paradigm, without altering their substance. Hence, we have compared these proposals against GPU state-of-the-art algorithms on real case datasets and the experimental results show their surprisingly good performance.

An explanation of their effectiveness has been provided thanks to a set of additional tests on synthetic images. The C-DRAG algorithm always outperforms the other CUDA-designed proposals, highlighting the feasibility of decision trees on GPU. The source code of described algorithms is available in [131], allowing anyone to reproduce and verify our claims.

However, two even better approaches, specifically designed for GPUs, will be presented in the next Section.

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. Experimental results on real cases and synthetically generated datasets demonstrate the superiority of the new proposals with respect to state-of-the-art, both on 2D and 3D scenarios.

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 [67]. 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 [52] for sequential algorithms. This allows to drastically reduce the amount of memory accesses, thus improving overall performance.

In previous work [63], the 2×2 blocks were applied to the iterative GPU algorithm presented in [61] and improved in [62]. However, the chosen algorithm was rather slow compared to other approaches [64, 66]. Moreover, the benefit introduced in [62] 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 [66] and [64]. 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 6.2.1. Anyway, their fundamental characteristics are summarized here to simplify the reading.

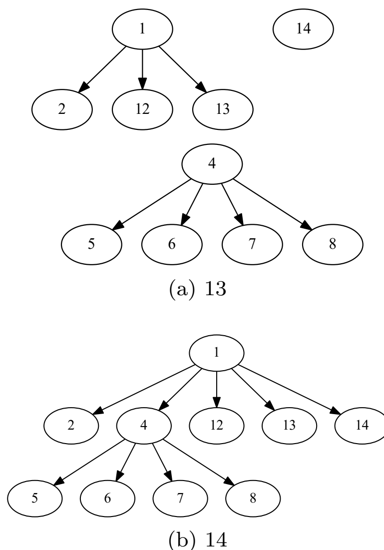


Figure 6.9: Change in the configuration of *union-find* trees operated by the thread working on pixel 14, and starting from the hypothetical provisional result of Fig. 6.10c. The thread checks its neighborhood mask in increasing raster index order. A `Union` is performed between the single-node tree 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.

The Union-Find algorithm

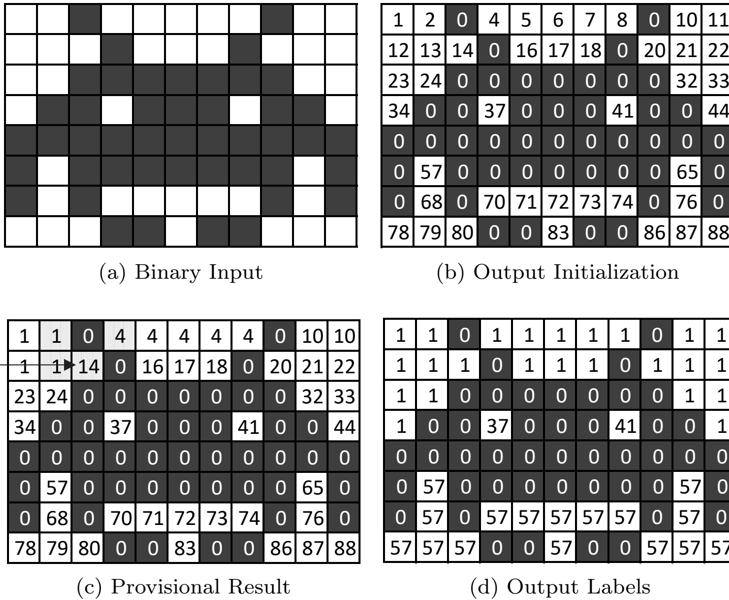
As said, UF is the first algorithm exploiting *union-find* on GPU. The original version of the algorithm employs 4-connectivity, but it can be easily extended to 8-connectivity [68]. The algorithm is based on three kernels: *Initialization*, *Merge* and *Compression*. An example of execution is depicted in Fig. 6.10. 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.

During *Merge*, each thread working on a foreground pixel checks its neighbors, and for every fore-

ground 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. 2.1a.

Fig. 6.9 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. 6.10c. In this example, the thread operating on pixel 14 performs a `Union` between the tree rooted in 14 and the



(a) Binary Input (b) Output Initialization

(c) Provisional Result (d) Output Labels

Figure 6.10: 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. 6.9. (d) is the labels image after the execution of the last kernel, *Compression*.

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 *Local Merge*. 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 [66] 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 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 [68] (see Section 6.2 for more details).

6.4.4 The Block-Based Approach

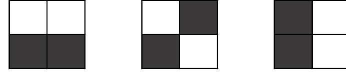


Figure 6.11: Examples of foreground pixels in a 2×2 block.

Grana *et al.* noticed in [52] 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. 6.11). 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.

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:

$$P \diamond Q \Leftrightarrow \exists\, p \in P, q \in Q \mid\ p \diamond q \tag{6.2}$$

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 [63].

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 8 and 9.

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.

(a) Provisional Labels

(b) Final Labels

Figure 6.12: 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.

In this case, *Merge* must be applied to blocks rather than to single pixels. The neighborhood mask used is depicted in Fig. 6.13. 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.* [63], 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.

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. 6.14. Initially, every bit is set to 0.

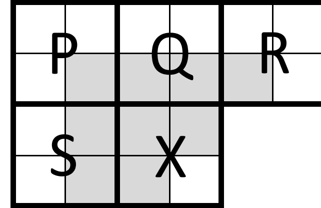


Figure 6.13: Neighborhood mask used by the two-dimensional block-based algorithms described in this Section. Central block is X, and the connectivity between it and its neighbors depends on the value of grey pixels.

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. 6.14b).

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 the `Compress` procedure defined in Section 3.2. 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. 6.12.

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

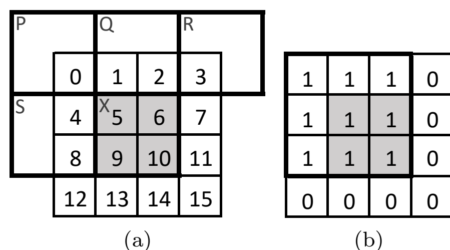


Figure 6.14: (a) shows how pixels in a 4×4 square centered on the X block are numerated. These numbers correspond to the pixel position in the associated bitset $\mathcal{BS}$. Bits 0, 1, 2, 3, 4, and 8 are used to record whether the corresponding pixel is to be checked for determining blocks connectivity 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.

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.

Table 6.6: 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`

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

Algorithm 8 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: $r \leftarrow (threadIdx.y + blockIdx.y \times blockDim.y) \times 2$
3: $c \leftarrow (threadIdx.x + blockIdx.x \times blockDim.x) \times 2$
4: $x_L \leftarrow r \times step_L + c$
5: $L[x_L] \leftarrow x_L$

6: **kernel** MERGE(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: $\mathcal{BS} \leftarrow 0$
12: **if** $I[x_I] = 1$ **then** $\mathcal{BS} \mid= 0x777$
13: **if** $I[x_I + 1] = 1$ **then** $\mathcal{BS} \mid= (0x777 << 1)$
14: **if** $I[x_I + step_I] = 1$ **then** $\mathcal{BS} \mid= (0x777 << 4)$
15: **if** $\mathcal{BS} > 0$ **then**
16: **if** `HasBit`$(\mathcal{BS}, 0)$ **and** $I[x_I - step_I - 1]$ **then**
17: `Union`$(L, x_L, x_L - 2 \times step_L - 2)$
18: **if** (`HasBit`$(\mathcal{BS}, 1)$ **and** $I[x_I - step_I]$) **or**
19: (`HasBit`$(\mathcal{BS}, 2)$ **and** $I[x_I - step_I + 1]$) **then**
20: `Union`$(L, x_L, x_L - 2 \times step_L)$
21: **if** `HasBit`$(\mathcal{BS}, 3)$ **and** $I[x_I - step_I + 2]$ **then**
22: `Union`$(L, x_L, x_L - 2 \times step_L + 2)$
23: **if** (`HasBit`$(\mathcal{BS}, 4)$ **and** $I[x_I - 1]$) **or**
24: (`HasBit`$(\mathcal{BS}, 8)$ **and** $I[x_I + step_I - 1]$) **then**
25: `Union`$(L, x_L, x_L - 2)$

again during *Reduction* and *FinalLabeling*, we save it in a bitmapped byte, called *information byte*. Its structure is explained in Table 6.6.

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.

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 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 9 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. 6.15.

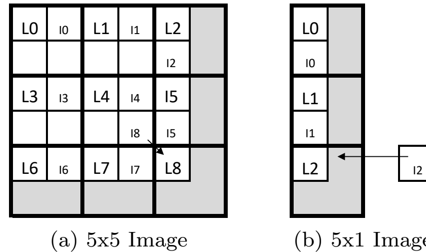
As for BUF, the *Compression* kernel flattens the *union-find* trees created in the previous phases to their roots. Two variations of this kernel exist, depending on the use of *InlineCompression*.

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 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*



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

Figure 6.15: 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.

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. 6.16. The remaining of the algorithm behaves similarly to its 2D counterpart, except for a slight optimization involving *Merge* and *FinalLabeling*.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31

32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47

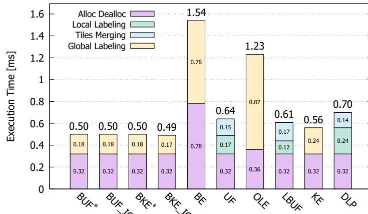
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63

Figure 6.16: 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.

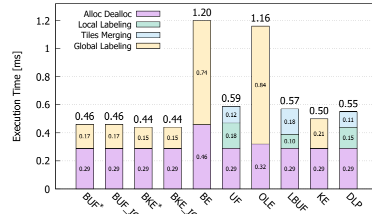
Kernel *FinalLabeling* is responsible for copying each block label in the corresponding foreground internal voxels, and assigning label 0 to background ones. In the absence of information about internal voxels, this procedure requires 8 memory readings, one for each internal voxel of a 2×2×2 cube.

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.

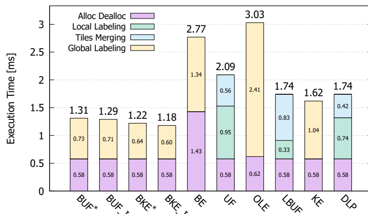
BKE is modified to suit 3D CCL in a similar way to BUF: 2×2×2 cubes are used in place of 2×2 squares, and the bitset that represents neighbor voxels is the same described above. The information byte used by the 2D variation becomes an information integer that requires 3 bytes in this case. In fact, it 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.



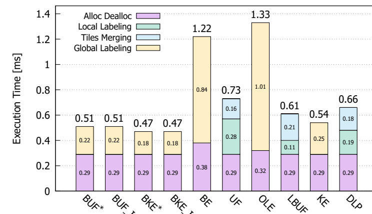
(a) 3DPeS



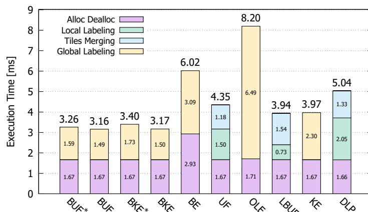
(b) Fingerprints



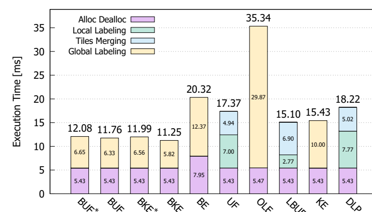
(c) Medical



(d) MIRflickr



(e) Tobacco800



(f) XDOCS

Figure 6.17: Average run-time with steps test on 2D datasets. Numbers are given in ms and our proposals are identified with *. Lower is better.

6.4.6 Comparative Evaluation

The proposed strategies are evaluated by comparing their performance with state-of-the-art algorithms.

Experimental results reported and discussed in this Section are ob-

tained running the YACCLAB benchmark on an Intel Core i7-4790 CPU (with 4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache), and using a Quadro K2200 NVIDIA GPU with Maxwell architecture, 640 CUDA cores and 4 GB of memory.

For convenience, the acronyms used to refer to the available algorithms are summarized here: BUF (Block-based Union-Find) and BKE (Block-based Komura Equivalence) are the two algorithms proposed in this Chapter. BE is the Block Equivalence algorithm proposed by Zavalishin *et al.* [63],

Table 6.7: Average run-time results in ms obtained under Windows (64 bit) OS with MSVC 19.15.26730 and NVCC V10.0.130 compilers using a Quadro K2200 NVIDIA GPU. The bold values represent the best performing CCL algorithm on a given dataset. Our proposals are identified with *. Lower is better.

	2D Images				
	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*
BUF*	0.512	0.441	2.161	1.313	0.495
BUF_IC*	0.508	0.440	2.112	1.299	0.494
BKE*	0.509	0.429	2.190	1.221	0.452
BKE_IC*	**0.501**	**0.427**	**2.073**	**1.186**	**0.449**
BE [63]	1.517	1.164	4.376	2.730	1.165
UF [64]	0.594	0.529	3.000	2.040	0.659
OLE [62]	1.211	1.128	5.358	3.013	1.281
LBUF [65]	0.573	0.509	2.776	1.699	0.541
KE [68]	0.568	0.481	2.717	1.622	0.526
DLP [69]	0.657	0.486	3.097	1.697	0.602

	2D Images		3D Volumes		
	XDOCS	*Tobacco800*	*Hilbert*	*Oasis*	*Mitochondria*
BUF*	12.088	3.268	2.119	6.792	65.829
BUF_IC*	11.764	**3.163**	2.119	6.802	**65.825**
BKE*	11.989	3.409	**2.097**	**6.728**	68.251
BKE_IC*	**11.253**	3.173	2.123	6.815	68.441
BE [63]	20.278	5.966	5.338	10.779	93.154
UF [64]	17.316	4.304	3.504	17.852	129.236
OLE [62]	35.242	8.173			
LBUF [65]	15.039	3.889			
KE [68]	15.432	3.978			
DLP [69]	18.172	5.002			

UF is Union-Find by Oliveira *et al.* [64], OLE is the Optimized version of Label Equivalence presented in [62] by Kalentev *et al.*, LBUF is the Line-Based Union-Find algorithm by Yonehara *et al.* [65], KE is the Komura Equivalence introduced in [66] by Komura, and DLP is the Distanceless Label Propagation algorithm by Cabaret *et al.* [69]. Finally IC identifies the variation described in Section 3.2 of BUF and BKE algorithms. The 3D version is available for all the algorithms except OLE, LBUF, KE, and DLP.

All the aforementioned algorithms have been implemented using CUDA 10.0 and compiled for x64 architectures, employing MSVC 19.15.26730 and NVCC V10.0.130 compilers with optimizations enabled.

The first experiment carried out is the comparison between algorithms in terms of average execution time over real datasets (Table 6.7). Our proposals, BUF and BKE, and all their variations outperform state-of-the-art algorithms on this test case. The best performing algorithm over 2D dataset is the IC variation of BKE, while, for what concerns 3D datasets, the base version provides better results.

As regards 2D, the speed-up between BKE_IC and KE, state-of-the-art competitor, varies from 1.1× on easy to label datasets (*3DPeS*) to 1.4× on datasets with a high number of complex connected components (*XDOCS*). In 3D test cases, the speed-up between BKE and BE ranges from 1.4× to 2.5×. Anyway, the performance of BUF and BKE are very close, and the same can be stated for their IC variations.

The *InlineCompression* optimization tries to reduce the number of memory accesses that an algorithm has to perform during the *Compression* phase, *i.e.*, the number of parent nodes a thread has to traverse to reach the root of the *union-find* equivalence tree. To achieve this goal an algorithm has to perform additional write operations that are of course expensive. Therefore, the benefit introduced by IC is valuable only when a convenient trade-off between saved readings and additional writings is achieved. This is linked to the complexity of the equivalence trees created and updated during *Initialization* and *Merge/Reduction* phases, which strictly depends on the nature of the input image like shape and dimension of the objects it contains, and on the order in which threads are executed. For this reasons, the definition of a break-even is very hard and cannot be done a priori.

Anyway, it is possible to observe that the use of IC always improves the performance of both BUF and BKE algorithms on 2D real cases datasets taken into account. In order to demonstrate this behavior, Table 6.8

is provided. In Table 6.8a the average number of memory accesses used by BUF and BUF_IC in the *Compression* kernel is provided for the *Fingerprints*, *Medical*, *Tobacco800* and *XDOCS* datasets. This table demonstrates the strict correlation between the difference of memory reads and the speed-up of BUF_IC algorithm w.r.t. BUF. A similar consideration can be drawn for BKE (Table 6.8b). As regards 3D datasets, instead, the use of IC may slightly increase the total execution time. In these test cases the *union-find* trees tend to be short, and hardly any memory reads are saved by this optimization. This has a greater effect on the BKE algorithm since it performs the *Compression* kernel twice.

To better investigate the algorithms behavior, Fig. 6.17 and Fig. 6.18 are also reported for 2D and 3D datasets, respectively. In these figures, bar charts report separately the time needed for allocating data structures from the time 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 Hamlet dataset has been omitted from Fig. 6.17 for space constraints. Results are very close to those of the other document datasets, *i.e.*, Tobacco800 and XDOCS.

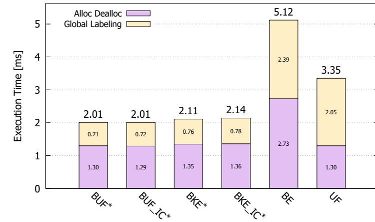
Focusing on Fig. 6.17, the allocation time is the same for each strategy, but for BE and OLE. Indeed, all the algorithms must only allocate memory for the output image.

Table 6.8: Effects of *InlineCompression* on the *Compression* kernel of BUF (a) and BKE (b), in terms of average amount of memory reads.

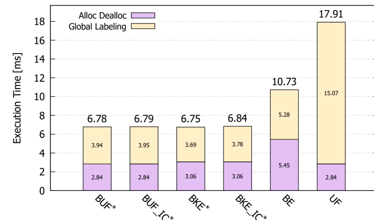
		BUF	BUF_IC	*Diff*
(a)	*Fingerprints*	37 395	30 147	7248
	Medical	313 871	252 421	61 450
	Tobacco800	273 592	215 796	57 796
	XDOCS	2 021 381	1 496 108	525 273
		BKE	BKE_IC	*Diff*
(b)	*Fingerprints*	23 868	21 130	2738
	Medical	149 334	145 627	3707
	Tobacco800	133 998	130 053	3945
	XDOCS	1 042 158	975 209	66 948

OLE, that is an iterative algorithm, requires an additional byte to check whether the convergence has been reached or not. This costs $0.03-0.04$ ms independently of the input image size. On the other hand, BE always requires a higher allocation time, since it relies on additional matrices to store equivalences between blocks and their labels. Obviously, this additional time is data dependent.

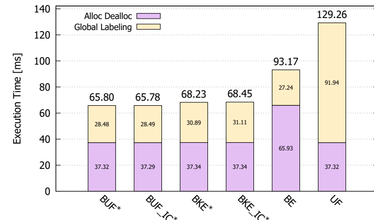
The charts show that the allocation and deallocation of device memory require a significant amount of the total execution time: that is why allocating or freeing global memory in performance-sensitive code should be done only when strictly necessary [94]. Anyway, a CCL algorithm cannot avoid the allocation of the output image. Thus, optimization can be applied only to the core part of the labeling procedure. In the light of these considerations, if we remove this fixed allocation cost, the speed-up of BKE_IC over KE moves from $\times 1.1-\times 1.4$ to $\times 1.4-\times 1.7$ on 2D datasets.



(a) Hilbert



(b) Oasis



(c) Mitochondria

Figure 6.18: Average run-time with steps test on 3D datasets. Numbers are given in ms and our proposals are identified with *. Lower is better.

The execution time of the OLE global labeling is always the worst, given that it requires many iterations over the input image to update the output one until convergence.

The block scan approach introduced by BE allows to highly reduce the operations required by OLE, thus reducing the global labeling time at the expense of allocation step.

Generally, UF has better perfomance than BE on 2D datasets. This is especially true on 3DPes, MIRflickr and Fingerprints, where many connected components are large and irregularly shaped. In such circumstances, BE requires more iterations to recognize an object as a single component. In the cases of components with simpler shapes, the performances of BE Global Labeling are comparable or better than those of UF. Anyway, the allocation time is too high for BE to be competitive in these scenarios.

The Line-Based variation of UF (LBUF) allows to simplify the logic of Local Labeling, reducing the neighborhood of a pixel to be checked during the *Local Merge* procedure. This algorithm moves the complexity to Tiles Merging, but always improves performance w.r.t. UF on 2D datasets. DLP tries to put together positive aspects of both UF and LE, but only in few cases is able to outperform UF. KE, thanks to an optimized *Initialization* kernel w.r.t. UF (Section 6.4.3), always improves its performance.

With our approaches we are able to combine the strengths of different strategies and benefit from the use of a block-based algorithm without increasing the amount of required memory, thus obtaining the lowest execution times.

Similar considerations can be drawn for 3D experiments (Fig. 6.18). On Mitochondria, the Global Scan of BE is faster than those of our proposals. This can be again explained considering the nature of the dataset, which is mostly composed of convex objects. Nevertheless, the massive memory usage makes the total execution time of BE higher than those of the proposed algorithms. That said, there are cases in which it could be fair to compare algorithms without considering memory allocation: in an embedded system in which images are always captured with the same size, for example, it could be realistic to allocate memory only once. In such scenarios, BE may be the best choice for data similar to Mitochondria.

Table 6.9: Average number of iterations required by the OLE algorithm on images of increasing density at 1-granularity level.

density (%)	0	10	20	30	40	50	60	70	80	90	100
iterations	1.0	4.0	5.0	5.0	7.2	5.0	5.0	4.0	3.9	3.0	2.0

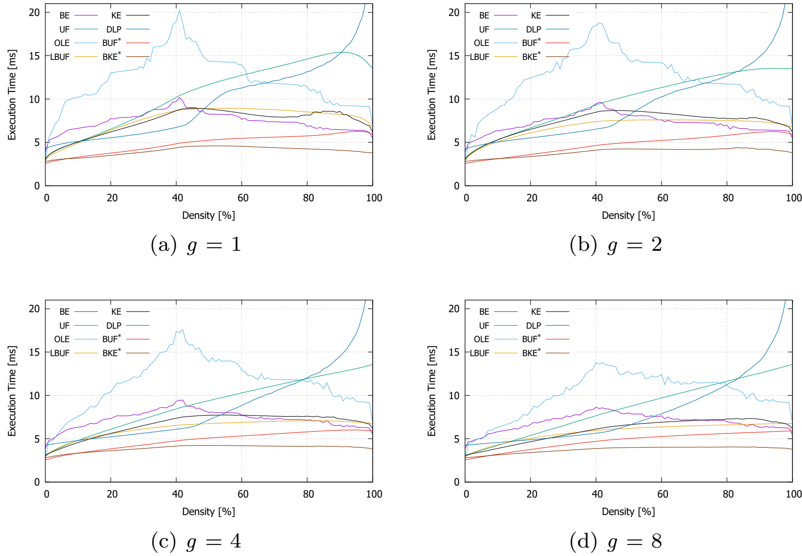


Figure 6.19: Two-dimensional tests on randomly generated images. Numbers are given in ms and our proposals are identified with *. Lower is better.

Following a common approach in literature [52, 53, 57, 134], additional tests have been performed on images with increasing foreground density and granularity (Fig. 6.19 and Fig. 6.21), in order to highlight strengths and weaknesses of the algorithms. To make the charts more readable the IC version of both BKE and BUF has been omitted.

Focusing on 2D datasets, it can be said that, independently of the pixel granularity, OLE has an increasing trend in the execution time up to 40% of foreground density, and then a decreasing one after this value. This behavior is strictly linked to the iterative nature of the algorithm. Indeed, the number of iterations required by the labeling procedure to converge reaches the highest value when foreground density is about 40% (Table 6.9). BE has a similar behavior, albeit with better performance.

The execution time of UF grows with foreground density. The reason is that each pixel thread has to perform one `Union` for each connected

neighbor, and the number of those pixels is linked to image density. The more `Union` there are, the more memory accesses and atomic operations are performed.

As shown in Fig. 6.19, KE and LBUF have a behavior equal to UF with densities up to 20%. Then, with larger connected components, KE *Initialization* is able to create shallower equivalence trees, and LBUF is able to create long lines of equivalent labels, again reducing the tree traversals required later. While using different strategies, KE and LBUF have similar trends with respect to pixels density, both largely improving on UF at high densities.

BUF has a similar trend to UF, since it inherits its basic behavior. The adoption of a block-based approach, anyway, allows to decrease the amount of atomic operations and memory accesses, drastically reducing the total execution time. At 80% density and above, the high number ot `Union` operations makes BUF slower than BE. Anyway, such density values are rather uncommon in real cases.

The execution times of BUF and BKE are very similar for low density images. Then, BKE has a decreasing trend after 40%. In fact, after that value large connected components start to appear, and the effect of the improved *Initialization* can be especially appreciated. The relationships between the algorithms remain the same as the granularity grows, but curves tend to be flatter.

For the sake of completeness, the results of tests over randomly generated three-dimensional volumes are reported as well, in Fig. 6.21. The same considerations drawn for the two-dimensional case can be applied.

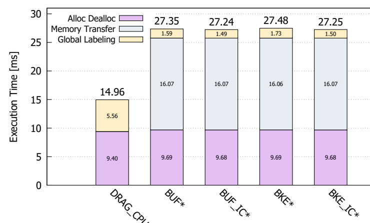
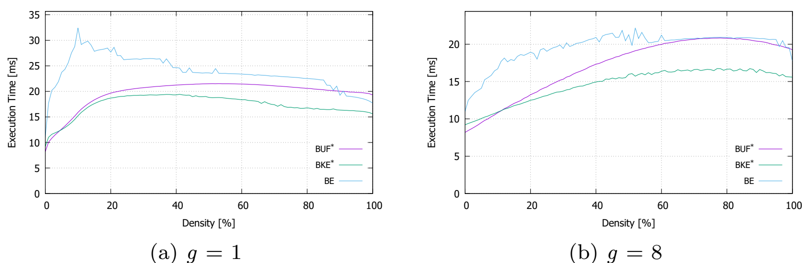


Figure 6.20: Comparison of the proposed algorithms with a state-of-the-art CPU-based CCL algorithm (DRAG) on Tobacco800 dataset. Numbers are given in ms and our proposals are identified with *. Lower is better.

Finally, Fig. 6.20 compares the proposed strategies to a state-of-the-art CPU-based CCL algorithm (DRAG [46]) on Tobacco800 dataset. In this scenario, differently from the other results reported in the manuscript, both

the input and the output images are in the host memory. Therefore, the elapsed time of the GPU algorithms includes the allocation/deallocation of GPU data structures, the allocation of the output image in CPU, and the data transfers between host and device memory. On the other hand, the CPU algorithm includes only the allocation of the output image and the required data structures. When considering only the Global Labeling, GPU algorithms have a speed-up between ×3.2 and ×3.7. Anyway, given the extremely high transfer time between host and device, a CLL GPU algorithm is preferable to a CPU one only as part of a GPU pipeline.



(a) $g = 1$ (b) $g = 8$

Figure 6.21: Three-dimensional tests on randomly generated images. Numbers are given in ms and our proposals are identified with *. Lower is better.

6.4.7 Conclusion

In this Section, the problem of GPU-based Connected Components Labeling in binary images and volumes has been addressed. Two new algorithms have been proposed, Block-based Union-Find (BUF) and Block-based Komura Equivalence (BKE), which have been obtained by combining existing strategies with a block-based approach, to considerably reduce the number of memory accesses and consequently improve time performance.

Experiments on a wide selection of both real case and synthetically generated datasets confirm that our proposals represent the state-of-the-art for GPU-based connected components labeling. The datasets cover most of the fields where CCL is commonly used, and allow to evaluate the correlation of performance to specific characteristics of the input. Among

the two proposals, BKE demonstrated superior performance in every test case, except for images with very low density. In fact, on random images, BUF has better performance than BKE for density below $5-10$, depending on the granularity.

Chapter 7

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

In this Chapter we present 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, 3D, CPU and GPU variants—, 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. Anyway, they are here summarized for easier reading of this Chapter.

7.1 Introduction

In this Section, 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.

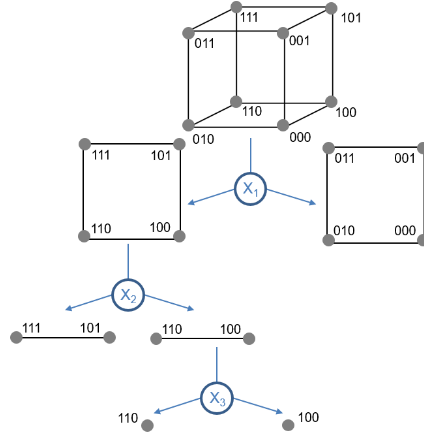


Figure 7.1: Example of 3-dimensional hypercube partitioning.

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 and parallel GPU-based scenarios as presented in Section 7.3. The source-code of the framework, and the complete documentation are available in [138].

7.2 GRAPHGEN

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* (Fig. 2.1). Different algorithms will obviously perform diverse actions based on the neighborhood, but even when the task is the same, algorithmic solutions may differ in the 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 step of an algorithm, only the pixels which are effectively required, while reducing the corresponding machine code.

7.2.1 Modelling Algorithms with Decision Tables

The class of algorithms that GRAPHGEN deals with can be described with a *command execution metaphor* [52]. This concept has already been introduced in the previous Chapters, 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 $\mathcal{DT} : R \to \mathcal{P}(A) \setminus \{\emptyset\}$, 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. 7.2). 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 [128] with a dynamic programming approach and extended in [60] to OR-decision tables.

Conditions	x	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	condition outcomes
	p	-	0	1	0	0	0	1	1	1	0	0	0	1	1	1	0	1	
	q	-	0	0	1	0	0	1	0	0	1	1	0	1	1	0	1	1	
	r	-	0	0	0	1	0	0	1	0	1	0	1	1	0	1	1	1	
	s	-	0	0	0	0	1	0	0	1	0	1	1	0	1	1	1	1	
Actions	no op.	*1*																	action entries
	new		*1*																
	p			*1*				1		1				1	1			1	
	q				*1*			*1*			*1*	*1*		*1*	*1*		*1*	*1*	
	r					*1*					1			1			1	1	
	s						*1*			*1*		1			1		1	1	
	p + r								*1*							1			
	r + s												*1*			*1*			

Figure 7.2: 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. 7.1) (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 7.2.1.1 (Decision Tree)**.** Given a $\mathcal{DT}$ and a k-cube K, a Decision Tree for K is a binary tree T where:

1. each leaf ℓ corresponds to a k-cube, denoted by $K_\ell \subseteq K$, and the set of K_ℓ is a partition of K;
2. each leaf ℓ has a non empty set of actions A_{K_ℓ}, associated to cube K_ℓ by $\mathcal{DT}$;
3. each internal node is labeled with an index $i \in D_K$ and is weighted

by w_i (which represents the cost of testing the i-th condition), with left and right outgoing edges labeled respectively with 0 and 1;
4. root to leaf paths uniquely identify cubes associated to leaves by means of nodes and edges' labels.

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

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:

$$gain(T) = \sum_{\ell \in \mathcal{L}} \left(P_{K_\ell} \sum_{i \in D_{K_\ell}} w_i \right) \tag{7.1}$$

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:

$$Gain_S = \max_{i \in D_S}(Gain_{S_{i,0}} + Gain_{S_{i,1}} + \delta w_i P_S) \tag{7.2}$$

Where δ equals 0 if $A_S = \emptyset$ and 1 otherwise. The procedure stops when $n = k$, and the optimal decision tree is constructed by tracing back through the merges that produce the maximum gain at each n-cube, until a cube S with $A_S \neq \emptyset$ is found, which is a leaf. Grana *et al.* proved the correctness of the algorithm in [60].

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 7.1: Example of YAML configuration file which defines the SAUF CCL algorithm [54] 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 7.1), GRAPHGEN is able to generate the optimal decision tree and the associated source code.

7.2.2 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 5, 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. 7.3, 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 [139]. He *et al.* [48] 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 [121] (Section 5.1) we proposed a paradigm to leverage already seen pixels, which combines prediction with decision trees. They noticed that

the knowledge of pixels checked in the previous step could result in a simplification of the DTree for the current pixel. A reduced DTree can be computed for each possible set of known pixels, and then the trees can be connected to generate a single forest, which drives the execution of the algorithm.

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

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 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

p_9^X	p_2^X p_9^Y	p_3^X p_2^Y	p_3^Y
p_8^X	p_1^X p_8^Y	p_4^X p_1^Y	p_4^Y
p_7^X	p_6^X p_7^Y	p_5^X p_6^Y	p_5^Y

Figure 7.3: Unitary horizontal shift for the 3×3 mask during image scan. Pixels named with "X" were inside the mask in the previous iteration while pixels named with "Y" are currently inside.

associated to the leaf, and only reduced DTrees are used until the end of the row.

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.

7.2.3 From Trees to DRAGs

The ODT generated in the first step of GRAPHGEN optimization procedure (Sec. 7.2.1) can contain identical or equivalent subtrees, as we noticed in [46]. 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 5.2.

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 anticipated above, the conversion from tree to DRAG has the benefit of reducing the machine code size. The purpose is to make better use of the instruction cache, obtaining a more efficient code compression than that achievable by a compiler, which can merge identical pieces of code but cannot exploit equivalence between subtrees. 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 [140]. 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.

7.3 Three Showcase Applications

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

and CUDA 10.0.130 compiler (x64) with optimizations enabled.

All discussed algorithms (unless noted otherwise) use decision trees or forests generated by GRAPHGEN. They have been proved to be correct, *i.e.* the output result is exactly the one required by the given task. The experiments are performed on the YACCLAB dataset (Section 4.2).

7.3.1 Connected Components Labeling

Connected Components Labeling aims at transforming an input binary image into a symbolic one, in which all pixels of the same object (connected component) are given the same label. The task has been described in detail in the previous Chapters, so we will skip directly to experimental results.

For comparing the GRAPHGEN generated algorithms with existing ones, the open-source benchmarking system YACCLAB [129, 132] has been used (see Section 4.3). 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 7.1). 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 preserving the benefit of the block-based approach (ASU=1.425). Since the Tagliatelle tree is larger than that of BBDT, applying compression (Spaghetti) yields a bigger improvement (ASU=1.492). As explained in Section 7.2.1, 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 7.1: 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.

7.3.2 Image Skeletonization

Image skeletonization, another fundamental algorithm used in many computer vision and image processing tasks, aims at providing an approximate and compact representation of the objects inside images, reducing them to one pixel wide “skeletons”. A common strategy to obtain it, called *thinning*, iteratively removes the outermost layers of connected components [70].

Many thinning algorithms belong to the class of parallel thinning algorithms [141]: 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 [74] 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 [75]. Furthermore, *(iii)* Guo and Hall [78] 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 [25, 26, 27] 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 10).

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 10 Two subiteration thinning algorithm

1: **function** ITERATION(I, O, k)
2: $O \leftarrow I$
3: $changed \leftarrow$ **false**
4: **for all** $p \in \mathcal{L}(I)$ **do**
5: **if** $I(p) = 1$ **then**
6: **if** SHOULD_REMOVE(I, p, k) **then**
7: $O(p) \leftarrow 0$
8: $changed \leftarrow$ **true**
9: **return** $changed$

10: **procedure** THINNING(I, O)
11: **repeat**
12: $changed_0 \leftarrow$ ITERATION($I, O, 0$)
13: $changed_1 \leftarrow$ ITERATION($O, I, 1$)
14: **until** $\neg changed_0 \wedge \neg changed_1$

Algorithm 11 and Algorithm 12 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 11 Removal logic functions for ZhangSuen and ChenHsu algorithms

```
 1: function A(P)
 2:     return (¬P2 ∧ P3) + (¬P3 ∧ P4) + (¬P4 ∧ P5) + (¬P5 ∧ P6)+
 3:            (¬P6 ∧ P7) + (¬P7 ∧ P8) + (¬P8 ∧ P9) + (¬P9 ∧ P2)
 4: function B(P)
 5:     return P2 + P3 + P4 + P5 + P6 + P7 + P8 + P9

 6: function ZS_SHOULD_REMOVE(I, p, k)
 7:     P ← I(N(p))
 8:     if k = 0 then
 9:         c ← P2 ∧ P4 ∧ P6;
10:         d ← P4 ∧ P6 ∧ P8;
11:     else
12:         c ← P2 ∧ P4 ∧ P8;
13:         d ← P2 ∧ P6 ∧ P8;
14:     return (A(P) = 1) ∧ (2 ≤ B(P) ≤ 6) ∧ ¬c ∧ ¬d

15: function CH_SHOULD_REMOVE(I, p, k)
16:     P ← I(N(p))
17:     if k = 0 then
18:         c ← P2 ∧ P4 ∧ P6;
19:         d ← P4 ∧ P6 ∧ P8;
20:         f ← P2 ∧ P4 ∧ ¬P6¬P7¬P8
21:         g ← P4 ∧ P6 ∧ ¬P2¬P8¬P9
22:     else
23:         c ← P2 ∧ P4 ∧ P8;
24:         d ← P2 ∧ P6 ∧ P8;
25:         f ← P2 ∧ P8 ∧ ¬P4¬P5¬P6
26:         g ← P6 ∧ P8 ∧ ¬P2¬P3¬P4
27:     return (2 ≤ B(P) ≤ 7) ∧ ((A(P) = 1) ∧ ¬c ∧ ¬d)∨
28:            (A(P) = 2) ∧ (f ∨ g))
```

Algorithm 12 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 [75] 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.

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. 2.2a) have been optimized with GRAPHGEN and compared with the open-source framework THeBE (THinning evaluation BEnchmark) [142]. THeBE is an extension of YACCLAB that we proposed in [143] 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. 7.4.

When comparing the various implementations within each algorithm

(Table 7.2), 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 7.2: 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**

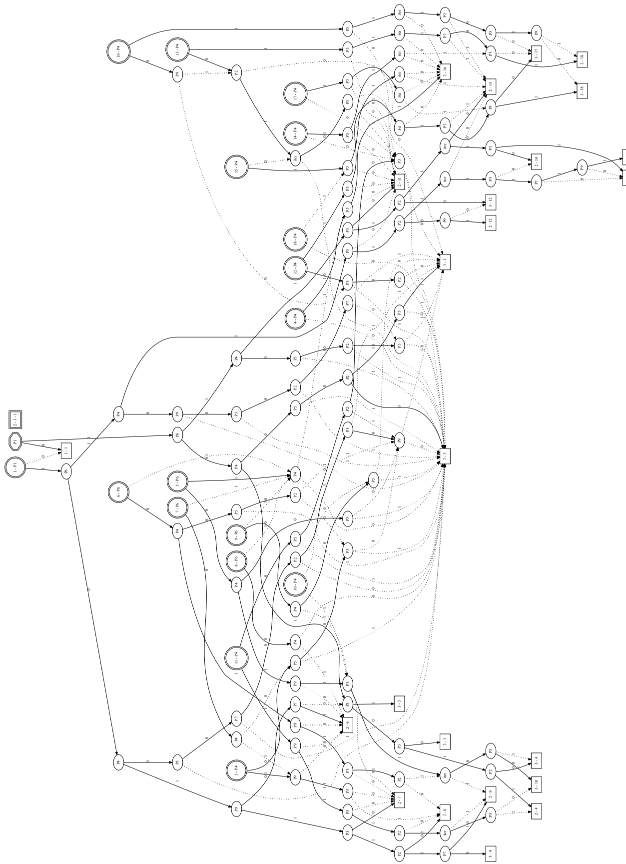


Figure 7.4: GRAPHGEN-generated DRAG for Zhang and Suen algorithm. The notation used is the same described in Section 5.3: the octagonal shaped root is the starting node for the first pixel in a line, double circles are roots from which the algorithm will restart after reaching a leaf (the first number 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.

7.3.3 Contours Extraction

In a binary image, a contour is a sequence of foreground pixels that separates a connected component from the background. Several methods have been proposed for the representation of a contour, among which the chain code is one of the most common. The first chain code variation was proposed by Freeman [81]. 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.

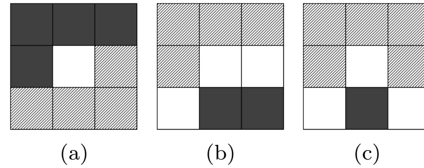


Figure 7.5: Max point templates. Dark squares are background, white squares are foreground and squares pattered with diagonal lines represent the concept of indifference. All outer max points correspond to template (a), while inner max points can either match (b) or (c).

Cederberg [91] 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. 7.5. 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. 7.6. 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.

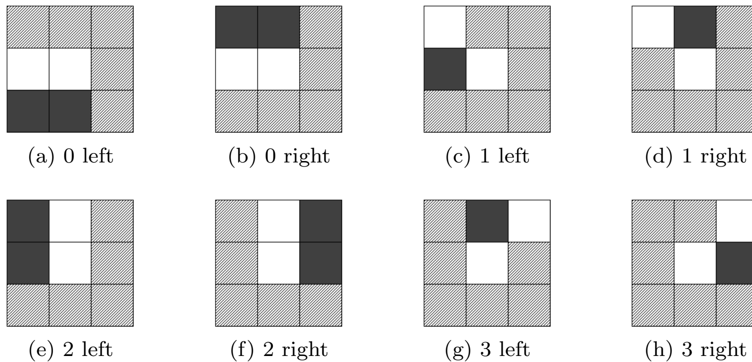


Figure 7.6: Chain link templates. Dark squares are background, white squares are foreground and squares patterned with diagonal lines represent the concept of indifference.

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. 7.7. In Fig. 7.7a, 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. 7.7c, the scan reaches a min point that links together M_0 left chain and M_1 right chain. Fig. 7.7d 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 7.2.

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

Listing 7.2: Definition of the max point structure in *C++*.

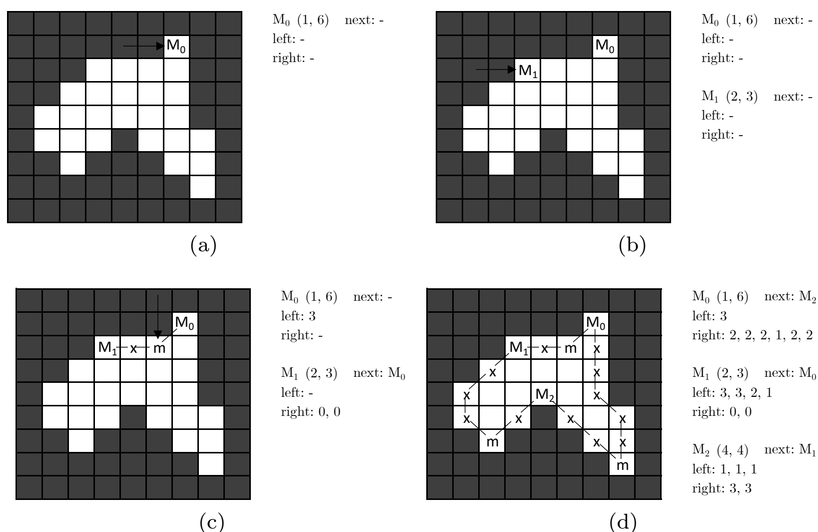


Figure 7.7: Example of execution of the RC-code extraction. The algorithm proceeds in raster scan, and in (a), (b) and (c) the arrow points to the pixel currently analyzed. Symbols M, m and x respectively represent max points, min points and chain links. To the left, max points are listed, along with 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.

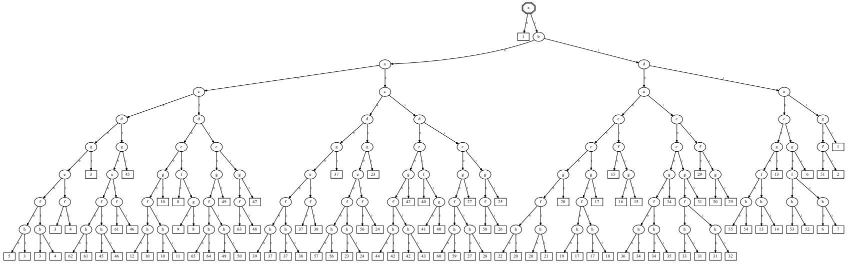


Figure 7.8: Binary decision tree which provides the Chain Code pixel status with minimal load/store operations. Best viewed online.

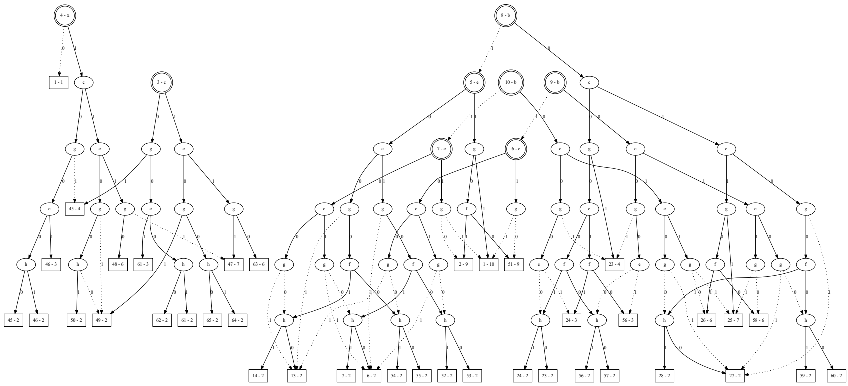
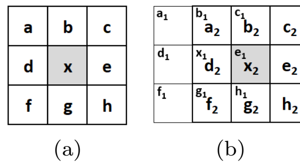


Figure 7.9: Part of the binary DRAG which provides the Chain Code pixel status with minimal load/store operations. Best viewed online.

When computing the RC-code, is it sufficient to look at the mask of Fig. 7.10 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. 7.8 and Fig. 7.9.



(a) (b)

Figure 7.10: (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 [144]. The reference algorithm is the one implemented by OpenCV 3.4.7 [145], which uses an extremely optimized contour following approach, while the algorithm proposed by Cederberg [91] has been implemented in multiple variants: using lookup tables with and without state prediction (LUT_PRED and LUT),

Table 7.3: 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 7.3), 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).

7.3.4 What About 3D and GPUs?

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

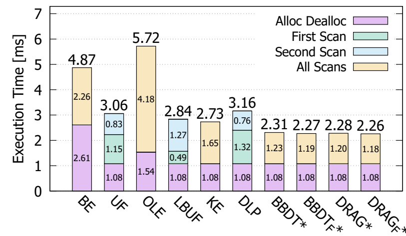


Figure 7.11: Average run-time experimental results on 2D CCL algorithms on GPU on the *Hamlet* dataset in milliseconds. The star identifies novel variations on previous algorithms generated thanks to GRAPHGEN. Lower is better.

With these optimizations, it can be observed that the BBDT and DRAG implementations perform better than previous state-of-the-art algorithms for CCL on GPUs [62, 63, 64, 65, 66, 69] (Fig. 7.11).

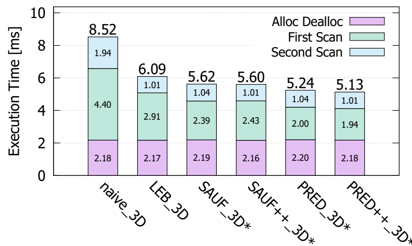
Since DRAG uses compression over BBDT, a slight advantage in run-time can be observed. Using image frequencies, DRAG_F achieves the best run-time over all algorithms, 18% faster than KE, the state-of-the-art approach.

The high amount of *if*- and *goto*-statements in the generated decision tree do not allow for efficient massively parallel and synchronized thread execution, but we have to consider the fact that branches depend on the pixel distribution in the mask. Neighboring pixels get processed concurrently, but they are not *i.i.d.*, and even partially overlapped and definitely

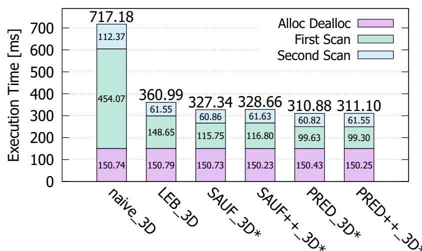
correlated.

Thus, it is likely that most of the time, threads will traverse the same path through the tree/DRAG without causing any divergence. A similar behavior can be observed on other datasets: these results are available in Appendix B.

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 $10^{24.68} TB$.



(a) Hilbert



(b) Mitochondria

Figure 7.12: 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.

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 imple-

mentation 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.* [146] (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. 7.12 shows the effectiveness of our proposal also in this case.

7.4 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.

Chapter 8

Conclusion

The goal of the research activities I carried out during my Ph.D. was the optimization of binary image processing algorithms. That goal has been tackled by following three main themes which motivate all the work presented in this thesis.

First of all we collected a comprehensive dataset for comparing CCL algorithms and designed a portable open-source *C++* benchmarking framework to fairly compare different algorithms. This tool, called YACCLAB, allows any new improvement to be evaluated uniformly with respect to existing proposals and cover a lack in literature. After its first appearance in 2016, it has been used by many authors to compare the performance of novel proposals with state-of-the-art solutions [59, 147, 148, 149, 150, 151] thus setting a *de-facto* standard.

Thanks to the analysis carried out through YACCLAB we were able to design new paradigms to label connected components, significantly improving the state-of-the-art on both sequential and parallel environments. For what concerns the sequential environment, the designed approach called Spaghetti Labeling, is a combination of different strategies that we proposed during my PhD: automatic simplified trees generation that exploits state prediction, along with a solution to leverage all the savings obtainable at image edges (*i.e.* the removal of *if* statements) and a greedy algorithm designed to convert decision forests into DAGs, thus reducing the machine code size more than a compiler could ever do. Thanks to the minimization of the code footprint, the load/store and merge operations, and *if*

statements required by the labeling procedure, the final resulting approach showed superior performance beating the results obtained by all compared algorithms in all settings.

On the other hand, for what concerns parallel environments the contribution to the Image Processing community is twofold: we proved that it is possible to parallelize existing sequential solutions to fit CPU multithreading environments. Moreover, two novel algorithms specifically designed for GPUs have been proposed. The algorithms, called Block-based Union-Find (BUF) and Block-based Komura Equivalence (BKE), have been obtained by applying a block-based approach to existing strategies, thus being able to considerably reduce the number of memory accesses and consequently improve time performance. Experiments on a wide selection of both real case and synthetically generated datasets confirm that our proposals represent the state-of-the-art for GPU-based connected components labeling.

Finally, we introduce a novel framework, GRAPHGEN, that allows to automatically apply the most effective optimization strategies presented in this thesis or previously published 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, image skeletonization and contour tracing. Furthermore, implementations for usage in parallel GPU environments and for 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.

Publications and Achievements

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

Among all the others, the work about YACCLAB has been published in the "Journal of Real-Time Image Processing". The work on Spaghetti

labeling has resulted in a journal paper on "IEEE Transactions on Image Processing" and two conference papers. The effort on parallel algorithms has produced a journal paper on "IEEE Transcations on Parallel and Distributed Systems", as well, and many other conference papers. Some of the results presented, instead, are under revision in major conferences or journals, like the work on GRAPHGEN.

In Appendix A, we report 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] W.-Y. Chang and C.-C. Chiu, "An efficient scan algorithm for block-based connected component labeling," in *22nd Mediterranean Conference of Control and Automation (MED)*, pp. 1008–1013, IEEE, 2014. xii, 11, 28

[2] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017. 1, 2

[3] F. Pollastri, F. Bolelli, R. Paredes, and C. Grana, "Augmenting Data with GANs to Segment Melanoma Skin Lesions," in *Multimedia Tools and Applications Journal*, MTAP, Springer, 2019. 1, 2, 10

[4] 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

[5] 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

[6] L. Baraldi, C. Grana, and R. Cucchiara, "Hierarchical Boundary-Aware Neural Encoder for Video Captioning," in *2017 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1657–1666, 2017. 1

[7] F. Bolelli, L. Baraldi, and C. Grana, "A Hierarchical Quasi-Recurrent approach to Video Captioning," in *2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS)*, pp. 162–167, IEEE, 2018. 1

[8] B. Wang, L. Ma, W. Zhang, W. Jiang, J. Wang, and W. Liu, "Controllable Video Captioning with POS Sequence Guidance Based on Gated Fusion Network," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2641–2650, 2019. 1

[9] C.-Y. Wu, C. Feichtenhofer, H. Fan, K. He, P. Krahenbuhl, and R. Girshick, "Long-Term Feature Banks for Detailed Video Understanding," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 284–293, 2019. 1

[10] X. Yang, X. Yang, M.-Y. Liu, F. Xiao, L. S. Davis, and J. Kautz, "STEP: Spatio-Temporal Progressive Learning for Video Action Detection," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 264–272, 2019. 1

[11] A. K. Bhunia, A. Das, A. K. Bhunia, P. S. R. Kishore, and P. P. Roy, "Handwriting recognition in low-resource scripts using adversarial learning*," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4767–4776, 2019. 1

[12] T. Wilkinson, J. Lindstrom, and A. Brun, "Neural Ctrl-F: Segmentation-Free Query-By-String Word Spotting in Handwritten Manuscript Collections," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4433–4442, 2017. 1

[13] I. H. Laradji, N. Rostamzadeh, P. O. Pinheiro, D. Vazquez, and M. Schmidt, "Where are the Blobs: Counting by Localization with Point Supervision," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 547–562, 2018. 1, 2

[14] B. Li, W. Ouyang, L. Sheng, X. Zeng, and X. Wang, "GS3D: An Efficient 3D Object Detection Framework for Autonomous Driving," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1019–1028, 2019. 1

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

[16] A. Palazzi, D. Abati, F. Solera, R. Cucchiara, *et al.*, “Predicting the Driver's Focus of Attention: the DR (eye) VE Project,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 7, pp. 1720–1733, 2018. 1

[17] M. Fabbri, F. Lanzi, S. Calderara, A. Palazzi, R. Vezzani, and R. Cucchiara, “Learning to Detect and Track Visible and Occluded Body Joints in a Virtual World,” in *European Conference on Computer Vision (ECCV)*, 2018. 1

[18] E. Ristani, F. Solera, R. Zou, R. Cucchiara, and C. Tomasi, “Performance Measures and a Data Set for Multi-target, Multi-camera Tracking,” in *Computer Vision – ECCV 2016 Workshops*, pp. 17–35, Springer, 2016. 1

[19] C. Liu, F. Wan, W. Ke, Z. Xiao, Y. Yao, X. Zhang, and Q. Ye, “Orthogonal Decomposition Network for Pixel-Wise Binary Classification,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6064–6073, 2019. 1

[20] W. Chen and J. Hays, “SketchyGAN: Towards Diverse and Realistic Sketch to Image Synthesis,” in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9416–9425, 2018. 1, 2

[21] P. Tschandl, C. Rosendahl, and H. Kittler, “The HAM10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions,” *Scientific data*, vol. 5, 2018. 1

[22] G. Yang, X. Song, C. Huang, Z. Deng, J. Shi, and B. Zhou, “DrivingStereo: A Large-Scale Dataset for Stereo Matching in Autonomous Driving Scenarios,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 899–908, 2019. 1

[23] T. Falk, D. Mai, R. Bensch, Ö. Çiçek, A. Abdulkadir, Y. Marrakchi, A. Böhm, J. Deubner, Z. Jäckel, K. Seiwald, *et al.*, “U-Net:

deep learning for cell counting, detection, and morphometry," *Nature Methods*, vol. 16, no. 1, pp. 67—-70, 2019. 1

[24] F. Milletari, S.-A. Ahmadi, C. Kroll, A. Plate, V. Rozanski, J. Maiostre, J. Levin, O. Dietrich, B. Ertl-Wagner, K. Bötzel, *et al.*, "Hough-CNN: Deep learning for segmentation of deep brain regions in MRI and ultrasound," *Computer Vision and Image Understanding*, vol. 164, pp. 92–102, 2017. 2

[25] J. Khodadoust and A. M. Khodadoust, "Fingerprint indexing based on minutiae pairs and convex core point," *Pattern Recognition*, vol. 67, pp. 110–126, 2017. 2, 14, 15, 156

[26] F. Uslu and A. A. Bharath, "A recursive Bayesian approach to describe retinal vasculature geometry," *Pattern Recognition*, vol. 87, pp. 157–169, 2019. 2, 14, 15, 156

[27] X. Wang, X. Jiang, and J. Ren, "Blood vessel segmentation from fundus image by a cascade classification framework," *Pattern Recognition*, vol. 88, pp. 331–341, 2019. 2, 14, 15, 156

[28] S. Hannuna, M. Camplani, J. Hall, M. Mirmehdi, D. Damen, T. Burghardt, A. Paiement, and L. Tao, "DS-KCF: a real-time tracker for RGB-D data," *Journal of Real-Time Image Processing*, vol. 16, pp. 1439–1458, Oct 2019. 2

[29] W.-C. Tu, S. He, Q. Yang, and S.-Y. Chien, "Real-Time Salient Object Detection with a Minimum Spanning Tree," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016. 2

[30] A. Rosenfeld and J. L. Pfaltz, "Sequential Operations in Digital Picture Processing," *Journal of the ACM*, vol. 13, pp. 471–494, Oct. 1966. 10, 11

[31] 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. 10

[32] 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. 10

[33] 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. 10

[34] 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. 10

[35] 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. 10

[36] L. Canalini, F. Pollastri, F. Bolelli, M. Cancilla, S. Allegretti, and C. Grana, “Skin Lesion Segmentation Ensemble with Diverse Training Strategies,” in *Computer Analysis of Images and Patterns*, pp. 89–101, Springer, September 2019. 10

[37] 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. 10

[38] 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. 10

[39] 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. 10, 32, 110

[40] 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. 10

[41] T. Berka, "The Generalized Feed-forward Loop Motif: Definition, Detection and Statistical Significance," *Procedia Computer Science*, vol. 11, pp. 75–87, 2012. 10

[42] 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. 10

[43] 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. 10

[44] S. Allegretti, F. Bolelli, M. Cancilla, F. Pollastri, L. Canalini, and C. Grana, "How does Connected Components Labeling with Decision Trees perform on GPUs?," in *Computer Analysis of Images and Patterns*, pp. 39–51, Springer, September 2019. 10

[45] 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. 10

[46] 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, IEEE, 2018. 10, 77, 82, 84, 90, 142, 152

[47] 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. 10, 12

[48] 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. 10, 11, 12, 28, 30, 41, 44, 56, 57, 58, 62, 70, 73, 77, 90, 94, 150

[49] 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. 10, 72

[50] 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. 10

[51] R. Haralick, "Some Neighborhood Operators," in *Real-Time Parallel Computing*, pp. 11–35, Springer, 1981. 10

[52] 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. 10, 12, 13, 21, 30, 31, 40, 42, 44, 56, 57, 61, 64, 65, 70, 72, 75, 90, 94, 100, 104, 113, 123, 127, 141, 147

[53] 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, 11, 12, 23, 25, 40, 56, 70, 72, 113, 141

[54] 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. 10, 11, 36, 40, 44, 56, 62, 70, 90, 100, 150

[55] A. Rosenfeld and A. Kak, *Digital picture processing*. No. v. 1 in Computer science and applied mathematics, Academic Press, 1982. 11

[56] 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. 11, 41, 44, 61, 70

[57] 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. 11, 40, 113, 141

[58] 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. 11, 73, 202, 203, 204

[59] D. Zhang, H. Ma, and L. Pan, "A Gamma-signal-regulated Connected Components Labeling Algorithm," *Pattern Recognit.*, 2019. 11, 171

[60] 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. 12, 40, 56, 65, 66, 72, 76, 77, 148, 149

[61] 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. 13, 123

[62] 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. 13, 120, 123, 136, 137, 167

[63] 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. 13, 120, 123, 127, 129, 136, 167

[64] V. M. Oliveira and R. A. Lotufo, "A study on connected components labeling algorithms using GPUs," in *SIBGRAPI*, vol. 3, p. 4, 2010. 13, 24, 104, 120, 123, 136, 137, 167

[65] 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. 13, 24, 136, 137, 167

[66] 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. 14, 24, 106, 123, 126, 137, 167

[67] 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. 14, 52, 103, 109, 123

[68] 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)*, pp. 175–180, IEEE, 2018. 14, 24, 120, 124, 126, 136

[69] 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. 14, 24, 120, 136, 137, 167

[70] E. S. Deutsch, "Thinning Algorithms on Rectangular, Hexagonal, and Triangular Arrays," *Communications of the ACM*, vol. 15, no. 9, pp. 827–837, 1972. 14, 155

[71] G. Dinneen, "Programming pattern recognition," in *AFIPS '55 (Western): Proceedings of the March 1-3, 1955, western joint computer conference*, pp. 94–100, ACM, 1955. 14

[72] B. B. Chaudhuri and C. Adak, "An approach for detecting and cleaning of struck-out handwritten text," *Pattern Recognition*, vol. 61, pp. 282–294, 2017. 14

[73] S. He and L. Schomaker, "DeepOtsu: Document enhancement and binarization using iterative deep learning," *Pattern Recognition*, vol. 91, pp. 379–390, 2019. 14

[74] 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. 14, 156

[75] 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. 14, 156, 158

[76] C. M. Holt, A. Stewart, M. Clint, and R. H. Perrott, "An Improved Parallel Thinning Algorithm," *Communications of the ACM*, vol. 30, no. 2, pp. 156–160, 1987. 15

[77] R. W. Hall, "Fast Parallel Thinning Algorithms: Parallel Speed and Connectivity Preservation," *Communications of the ACM*, vol. 32, no. 1, pp. 124–131, 1989. 15

[78] Z. Guo and R. W. Hall, "Parallel Thinning with Two-Subiteration Algorithms," *Communications of the ACM*, vol. 32, no. 3, pp. 359–373, 1989. 15, 156

[79] H. Lü and P. S.-P. Wang, "A Comment on "A Fast Parallel Algorithm for Thinning Digital Patterns"," *Communications of the ACM*, vol. 29, no. 3, pp. 239–242, 1986. 15

[80] C. Grana and D. Borghesani, "Optimal Decision Tree Synthesis for Efficient Neighborhood Computation," in *AI*IA 2009: Emergent Perspectives in Artificial Intelligence*, (Reggio Emilia, Italy), pp. 92–101, 2009. 15

[81] H. Freeman, "On the Encoding of Arbitrary Geometric Configurations," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 260–268, June 1961. 16, 161

[82] K. Dunkelberger and O. Mitchell, "Contour tracing for precision measurement," in *Proceedings. 1985 IEEE International Conference on Robotics and Automation*, vol. 2, pp. 22–27, IEEE, 1985. 16

[83] G. R. Wilson, "Properties of contour codes," *IEE Proceedings-Vision, Image and Signal Processing*, vol. 144, no. 3, pp. 145–149, 1997. 16

[84] E. Bribiesca, "The Spirals of the Slope Chain Code," *Pattern Recognition*, 2019. 17

[85] E. Bribiesca, "A Geometric Structure for Two-Dimensional Shapes and Three-Dimensional Surfaces," *Pattern Recognition*, vol. 25, no. 5, pp. 483–496, 1992. 17

[86] S. Kundu and B. Ray, “An efficient chain code based face identification system for biometrics,” in *2015 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pp. 559–564, IEEE, 2015. 17

[87] J. W. McKee and J. Aggarwal, “Computer Recognition of Partial Views of Curved Objects,” *IEEE Transactions on Computers*, vol. C-26, no. 8, pp. 790–800, 1977. 17

[88] M. D. Levine, *Vision in man and machine.* McGraw-Hill College, 1985. 17

[89] Z. Shi and V. Govindaraju, “A chaincode based scheme for fingerprint feature extraction,” *Pattern Recognition Letters*, vol. 27, no. 5, pp. 462–468, 2006. 17

[90] B. Batchelor and B. Marlow, “Fast generation of chain code,” *IEE Proceedings E - Computers and Digital Techniques*, vol. 127, no. 4, pp. 143–147, 1980. 17

[91] 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. 17, 161, 166

[92] P. Zingaretti, M. Gasparroni, and L. Vecci, “Fast Chain Coding of Region Boundaries,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 4, pp. 407–415, 1998. 17

[93] 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. 21, 104

[94] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming.* Pearson Education, 2013. 24, 139

[95] S. Allegretti, F. Bolelli, M. Cancilla, and C. Grana, “A Block-Based Union-Find Algorithm to Label Connected Components on GPUs,” in *Image Analysis and Processing – ICIAP 2019*, pp. 271–281, Springer, 2019. 24

[96] B. A. Galler and M. J. Fisher, "An improved equivalence algorithm," *Communications of the ACM*, vol. 7, pp. 301–303, May 1964. 25

[97] 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. 25, 28

[98] 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. 25, 30, 56

[99] E. W. Dijkstra, *A discipline of programming.* Prentice-Hall Englewood Cliffs, N.J, 1976. 25, 40, 90

[100] M. Patwary, J. Blair, and F. Manne, "Experiments on Union-Find Algorithms for the Disjoint-Set Data Structure," *Experimental Algorithms*, pp. 411–423, 2010. 25

[101] A. Torralba and A. A. Efros, "Unbiased Look at Dataset Bias," in *CVPR 2011*, pp. 1521–1528, IEEE, 2011. 28

[102] L. Cabaret, L. Lacassagne, and D. Etiemble, "Parallel Light Speed Labeling: an efficient connected component algorithm for labeling and analysis on multi-core processors," *Journal of Real-Time Image Processing*, pp. 1–24, 2016. 28, 30, 39, 41, 44, 45, 53

[103] F. Bolelli, C. Grana, M. Cancilla, and S. Allegretti, "The YACCLAB Dataset." `http://imagelab.ing.unimore.it/yacclab`. Accessed on 2020-01-10. 30

[104] L. He, Y. Chao, K. Suzuki, and K. Wu, "Fast connected-component labeling," *Pattern Recognition*, vol. 42, no. 9, pp. 1977–1987, 2009. 30, 40

[105] L. Lacassagne and B. Zavidovique, "Light speed labeling: efficient connected component labeling on risc architectures," *Journal of Real-Time Image Processing*, vol. 6, no. 2, pp. 117–135, 2011. 30, 41, 61

[106] 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. 30, 41, 44, 56, 61

[107] P. Sutheebanjard and W. Premchaiswadi, "Efficient scan mask techniques for connected components labeling algorithm," *EURASIP Journal on Image and Video Processing*, vol. 2011, no. 1, pp. 1–20, 2011. 30

[108] 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. 31, 53

[109] 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. 31, 33, 52

[110] 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. 31, 110

[111] H. Zhao, Y. Fan, T. Zhang, and H. Sang, "Stripe-based connected components labelling," *Electronics letters*, vol. 46, no. 21, pp. 1434–1436, 2010. 31, 42, 44

[112] 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. 31, 110

[113] 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. 32, 110

[114] F. Bolelli, G. Borghi, and C. Grana, "Historical Handwritten Text Images Word Spotting Through Sliding Window Hog Features," in *Image Analysis and Processing – ICIAP 2017*, pp. 729–738, Springer, 2017. 32, 110

[115] 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. 32

[116] D. Maltoni, D. Maio, A. Jain, and S. Prabhakar, *Handbook of Fingerprint Recognition.* Springer Science & Business Media, 2009. 32, 110

[117] J. Sauvola and M. Pietikäinen, "Adaptive document image binarization," *Pattern recognition*, vol. 33, no. 2, pp. 225–236, 2000. 32

[118] 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. 32, 110

[119] L. Lacassagne and B. Zavidovique, "Light Speed Labeling for RISC architectures.," in *16th IEEE International Conference on Image Processing (ICIP)*, pp. 3245–3248, 2009. 41

[120] L. Di Stefano and A. Bulgarelli, "A Simple and Efficient Connected Components Labeling Algorithm," in *Proceedings 10th International Conference on Image Analysis and Processing*, pp. 322–327, IEEE, 1999. 41, 44, 61

[121] 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. 41, 44, 70, 73, 77, 79, 90, 94, 150

[122] F. Bolelli, S. Allegretti, and C. Grana, "The YACCLAB 3D Dataset." `http://aimagelab.ing.unimore.it/files/YACCLAB_dataset3D.zip`. Accessed on 2019-03-21. 52

[123] 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. 52

[124] A. Lucchi, Y. Li, C. Becker, and P. Fua, "The Electron Microscopy Dataset." `https://cvlab.epfl.ch/data/data-em/`. Accessed on 2020-01-08. 52

[125] 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. 52

[126] S. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*. Springer Science & Business Media, 1996. 64

[127] L. J. Schutte, “Survey of Decision Tables as a Problem Statement Technique,” CSD-TR 80, Computer Science Department, Purdue University, 1973. 65, 75

[128] H. Schumacher and K. C. Sevcik, “The Synthetic Approach to Decision Table Conversion,” *Communications of the ACM*, vol. 19, pp. 343–351, June 1976. 65, 76, 148

[129] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, “Toward reliable experiments on the performance of Connected Components Labeling algorithms,” *Journal of Real-Time Image Processing*, pp. 1–16, 2018. 70, 74, 89, 90, 92, 94, 110, 118, 154

[130] 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. 73, 76

[131] F. Bolelli, C. Grana, M. Cancilla, and S. Allegretti, “The YACCLAB Benchmark.” https://github.com/prittt/YACCLAB. Accessed on 2020-01-10. 89, 96, 114, 122

[132] 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, ICPR, 2016. 89, 110, 118, 154

[133] F. Bolelli, M. Cancilla, L. Baraldi, and C. Grana, “Connected Components Labeling on DRAGs: Implementation and Reproducibility Notes,” in *Reproducible Research in Pattern Recognition*, pp. 89–93, Springer, 2018. 90

[134] L. Cabaret, L. Lacassagne, and D. Etiemble, “Parallel light speed labeling: An efficient connected component labeling algorithm for multi-core processors,” in *2015 IEEE International Conference on Image Processing (ICIP)*, pp. 3486–3489, IEEE, 2015. 98, 99, 141

[135] F. Bolelli, M. Cancilla, and C. Grana, "Two More Strategies to Speed Up Connected Components Labeling Algorithms," in *Image Analysis and Processing – ICIAP 2017*, pp. 48–58, Springer, 2017. 100

[136] J. Nickolls and W. J. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, no. 2, 2010. 115

[137] 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. 115

[138] F. Bolelli, C. Grana, and M. Soëcthing, "GRAPHGEN source-code and documentation." `https://github.com/prittt/graphgen`. Accessed on 2020-01-10. This is still a private repository because it is associated to a paper under review in a double blind conference. 146

[139] 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. 150

[140] 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. 153

[141] 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. 155

[142] "Source code of the THeBE benchmarking system." `https://github.com/prittt/THeBE`. Accessed on 2019-05-02. 158

[143] F. Bolelli and C. Grana, "Improving the Performance of Thinning Algorithms with Directed Rooted Acyclic Graphs," *Image Analysis and Processing – ICIAP 2019*, pp. 148–158, 2019. 158

[144] F. Bolelli, S. Allegretti, and C. Grana, "BACCA source-code." `https://github.com/prittt/bacca`. Accessed on 2020-01-10. This is still a private repository because it is associated to a paper under review in a double blind conference. 166

[145] 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. 166

[146] 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. 169

[147] T. Chabardès, P. Dokládal, and M. Bilodeau, "A labeling algorithm based on a forest of decision trees," *Journal of Real-Time Image Processing*, pp. 1–19, 2019. 171

[148] T. Chabardes, P. Dokládal, M. Faessel, and M. Bilodeau, "A parallel, O(N) algorithm for unbiased, thin watershed," in *2016 IEEE International Conference on Image Processing (ICIP)*, pp. 2569–2573, IEEE, 2016. 171

[149] J. Chen, K. Nonaka, H. Sankoh, R. Watanabe, H. Sabirin, and S. Naito, "Efficient Parallel Connected Component Labeling with a Coarse-to-fine Strategy," *IEEE Access*, 2018. 171

[150] 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. 171

[151] D. Ma, S. Liu, and Q. Liao, "Run-Based Connected Components Labeling Using Double-Row Scan," in *International Conference on Image and Graphics*, pp. 264–274, Springer, 2017. 171

Appendix A

List of Publications

This Section contains the list of research papers published during the Ph.D. period, as well as pre-prints which are 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. Some of the topics which fall under this last category are:

- indexing of historical document images;
- hierarchical approaches to video captioning;
- skin lesion segmentation and classification;

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] Costantino Grana, Federico Bolelli, Lorenzo Baraldi, and Roberto Vezzani. YACCLAB - Yet Another Connected Components Labeling Benchmark. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 3109–3114. Springer, December 2016.

[2] Costantino Grana, Lorenzo Baraldi, and Federico Bolelli. Optimized Connected Components Labeling with Pixel Prediction. In *Advanced Concepts for Intelligent Vision Systems*, pages 431–440. Springer, October 2016.

[3] Federico Bolelli. Indexing of Historical Document Images: Ad Hoc Dewarping Technique for Handwritten Text. In *Digital Libraries and Archives*. Springer, February 2017.

[4] Federico Bolelli, Guido Borghi, and Costantino Grana. Historical Handwritten Text Images Word Spotting through Sliding Window HOG Features. In *Image Analysis and Processing - ICIAP 2017*, pages 729–738. Springer, September 2017.

[5] Federico Bolelli, Michele Cancilla, and Costantino Grana. Two More Strategies to Speed Up Connected Components Labeling Algorithms. In *Image Analysis and Processing - ICIAP 2017*, pages 48–58. Springer, September 2017.

[6] Federico Bolelli, Lorenzo Baraldi, Michele Cancilla, and Costantino Grana. Connected Components Labeling on DRAGs. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 121–126. IEEE, August 2018.

[7] Federico Bolelli, Lorenzo Baraldi, and Costantino Grana. A Hierarchical Quasi-Recurrent approach to Video Captioning. In *2018 IEEE International Conference on Image Processing, Applications and Systems (IPAS)*, pages 162–167. IEEE, December 2018.

[8] Federico Bolelli, Guido Borghi, and Costantino Grana. XDOCS: an Application to Index Historical Documents. In *Digital Libraries and Multimedia Archives*, pages 151–162. Springer, January 2018.

[9] Federico Bolelli, Michele Cancilla, Lorenzo Baraldi, and Costantino Grana. Towards reliable experiments on the performance of Connected Components Labeling algorithms. *Journal of Real-Time Image Processing*, pages 1–16, February 2018.

[10] 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.

[11] Stefano Pini, Marcella Cornia, Federico Bolelli, Lorenzo Baraldi, and Rita Cucchiara. M-VAD Names: a Dataset for Video Captioning with

Naming. *Multimedia Tools and Applications Journal*, pages 14007–14027, May 2018.

[12] Federico Pollastri, Federico Bolelli, Roberto Paredes, and Costantino Grana. Improving Skin Lesion Segmentation with Generative Adversarial Networks. In *2018 IEEE 31st International Symposium on Computer-Based Medical Systems (CBMS)*, pages 442–443. IEEE, June 2018.

[13] Federico Bolelli, Michele Cancilla, Lorenzo Baraldi, and Costantino Grana. Connected Components Labeling on DRAGs: Implementation and Reproducibility Notes. In *Reproducible Research in Pattern Recognition*, pages 89–93. Springer, May 2019.

[14] Federico Bolelli and Costantino Grana. Improving the Performance of Thinning Algorithms with Directed Rooted Acyclic Graphs. In *Image Analysis and Processing - ICIAP 2019*, pages 148–158. Springer, September 2019.

[15] 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.

[16] 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.

[17] 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.

[18] Federico Pollastri, Federico Bolelli, Roberto Paredes, and Costantino Grana. Augmenting Data with GANs to Segment Melanoma Skin Lesions. *Multimedia Tools and Applications*, pages 1–18, May 2019.

[19] 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.

[20] Federico Bolelli, Stefano Allegretti, Maximilian Soëcthing, and Costantino Grana. One DAG to Rule Them All. In *Computer Vision – ECCV 2020*. Springer, 2020. *Under Review*.

[21] Stefano Allegretti, Federico Bolelli, and Costantino Grana. A Warp Speed Chain Code Algorithm Based on Binary Decision Trees. In *IEEE 4th International Conference on Imaging, Vision & Pattern Recognition (IVPR)*. IEEE, 2020. *Under Review*.

[22] 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.

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.

Toward Reliable Experiments on Algorithms Performance, 4

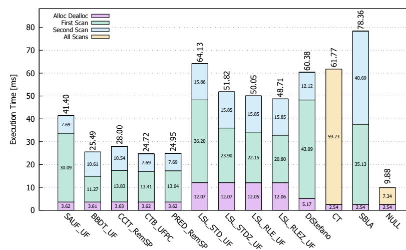


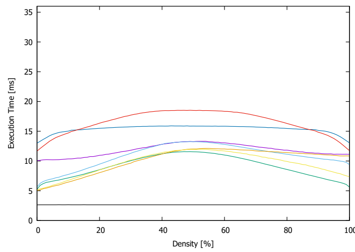
Figure B.1: Average run-time tests with steps in ms on an Intel Core i7-4770 *CPU* @ 3.40GHz running Linux with GCC 5.1.0 on the XDOCS dataset. Lower is better.

Table B.1: Average run-time results in ms obtained under Linux with GCC 5.1.0 compiler. The bold values represent the best labels solver for a specific CCL algorithm and dataset, the red ones point out the best algorithm for a given dataset. Lower is better.

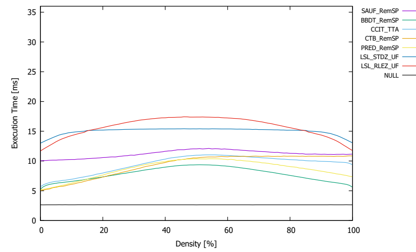
	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
SAUF_RemSP	**1.123**	0.404	6.832	**3.036**	**0.556**	**11.209**	**41.031**
SAUF_TTA	1.136	0.424	6.960	3.178	0.573	11.243	41.719
SAUF_UFPC	1.143	0.432	6.998	3.083	0.570	11.358	42.042
SAUF_UF	1.126	**0.400**	**6.827**	3.065	0.559	11.230	41.438
BBDT_RemSP	0.687	**0.330**	**4.376**	**1.985**	**0.391**	**6.477**	**24.632**
BBDT_TTA	0.699	0.339	4.512	2.047	0.398	6.612	25.561
BBDT_UFPC	0.688	0.335	4.480	2.003	0.395	6.571	25.223
BBDT_UF	**0.682**	0.333	4.412	2.037	0.395	6.500	24.882
CCIT_RemSP	0.801	0.378	4.915	2.303	**0.470**	7.447	27.100
CCIT_TTA	**0.759**	**0.367**	**4.747**	**2.267**	0.471	**6.975**	**26.648**
CCIT_UFPC	0.839	0.412	5.369	2.503	0.518	7.945	30.316
CCIT_UF	0.853	0.403	5.386	2.465	0.504	8.035	30.149
CTB_RemSP	0.669	**0.316**	**4.149**	2.097	0.458	**6.218**	24.501
CTB_TTA	0.675	0.317	4.175	**2.076**	**0.443**	6.219	24.576
CTB_UFPC	**0.669**	0.317	4.163	2.091	0.456	6.254	**24.452**
CTB_UF	0.673	0.325	4.194	2.078	0.452	6.263	24.537
PRED_RemSP	0.689	0.323	4.254	**1.972**	0.414	6.310	24.579
PRED_TTA	0.696	0.328	4.291	1.993	0.416	6.354	25.000
PRED_UFPC	0.691	0.330	4.289	2.003	0.422	6.361	24.853
PRED_UF	**0.684**	**0.323**	**4.219**	1.974	**0.413**	**6.259**	**24.469**
LSL_STD_TTA	2.100	0.689	11.640	5.638	0.949	18.400	64.207
LSL_STD_UF	**2.095**	**0.685**	**11.608**	**5.504**	**0.944**	**18.337**	**64.048**
LSL_STDZ_TTA	1.814	0.595	9.667	4.760	0.823	15.049	52.201
LSL_STDZ_UF	**1.796**	**0.583**	**9.511**	**4.572**	**0.811**	**14.794**	**51.313**
LSL_RLE_TTA	1.714	0.655	9.380	4.587	0.837	13.966	49.483
LSL_RLE_UF	**1.706**	**0.646**	**9.330**	**4.435**	**0.829**	**13.887**	**49.240**
LSL_RLEZ_TTA	1.713	0.655	9.385	4.587	0.837	13.983	49.543
LSL_RLEZ_UF	**1.696**	**0.637**	**9.221**	**4.396**	**0.818**	**13.785**	**48.687**
DiStefano	1.198	0.629	8.377	4.232	0.857	12.753	58.009
CT	1.435	1.027	11.447	5.371	1.073	15.804	66.645
SBLA	1.339	0.685	9.655	4.454	0.821	14.406	56.906
NULL	0.395	0.094	1.791	0.849	0.160	2.834	9.685

Table B.2: Average run-time results in ms obtained under Windows with GCC 5.1.0 compiler. The bold values represent the best labels solver for a specific CCL algorithm and dataset, the red ones point out the best algorithm for a given dataset. Lower is better.

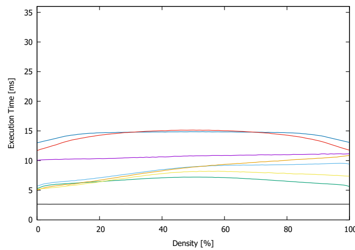
	3DPeS	*Fingerprints*	*Hamlet*	*Medical*	*MIRflickr*	*Tobacco800*	*XDOCS*
SAUF_RemSP	1.123	0.420	7.932	3.476	**0.578**	13.293	63.635
SAUF_TTA	1.139	0.429	7.945	3.525	0.590	**13.264**	64.126
SAUF_UFPC	1.141	0.442	8.029	3.485	0.589	13.425	**63.276**
SAUF_UF	**1.121**	**0.414**	**7.863**	**3.476**	0.581	13.265	63.607
BBDT_RemSP	0.723	0.362	5.490	2.311	0.422	8.756	42.290
BBDT_TTA	0.733	0.365	5.505	2.317	0.423	8.765	42.601
BBDT_UFPC	0.724	0.365	5.500	2.315	0.423	8.760	42.363
BBDT_UF	**0.609**	**0.321**	**4.715**	**1.977**	**0.370**	**7.484**	**36.727**
CCIT_RemSP	**0.845**	**0.400**	**6.307**	**2.806**	**0.508**	**10.126**	**50.326**
CCIT_TTA	0.894	0.418	6.665	3.009	0.540	10.712	53.974
CCIT_UFPC	0.855	0.424	6.431	2.899	0.545	10.196	51.185
CCIT_UF	0.853	0.416	6.389	2.851	0.529	10.157	50.337
CTB_RemSP	0.674	0.347	5.306	2.614	0.509	8.380	48.303
CTB_TTA	0.687	0.349	5.349	2.637	0.514	8.440	48.785
CTB_UFPC	**0.659**	0.346	5.208	2.606	0.516	**8.215**	47.846
CTB_UF	0.675	**0.335**	**5.203**	**2.436**	**0.468**	8.260	**44.461**
PRED_RemSP	0.693	0.349	5.363	2.411	0.446	8.436	43.561
PRED_TTA	0.707	0.355	5.431	2.436	0.450	8.505	44.071
PRED_UFPC	0.696	0.353	5.382	2.422	0.453	8.447	43.708
PRED_UF	**0.686**	**0.332**	**5.247**	**2.355**	**0.431**	**8.335**	**42.662**
LSL_STD_TTA	2.026	0.658	15.620	6.761	0.946	26.373	119.099
LSL_STD_UF	**2.011**	**0.653**	**15.567**	**6.743**	**0.940**	**26.299**	**118.757**
LSL_STDZ_TTA	1.745	0.565	13.590	5.867	0.823	23.043	104.320
LSL_STDZ_UF	**1.710**	**0.552**	**13.402**	**5.784**	**0.808**	**22.753**	**102.928**
LSL_RLE_TTA	1.635	**0.622**	**13.279**	**5.667**	**0.832**	**21.892**	**99.882**
LSL_RLE_UF	**1.627**	0.627	13.311	5.734	0.835	21.918	100.168
LSL_RLEZ_TTA	1.635	0.628	13.314	5.736	0.836	21.917	100.093
LSL_RLEZ_UF	**1.610**	**0.601**	**13.067**	**5.584**	**0.811**	**21.687**	**98.852**
DiStefano	0.820	0.555	6.879	3.619	0.738	10.293	72.852
CT	1.431	1.037	12.332	5.716	1.099	17.953	98.586
SBLA	1.263	0.682	10.089	4.569	0.817	15.494	80.206
NULL	0.364	0.096	2.452	1.086	0.163	4.397	21.378



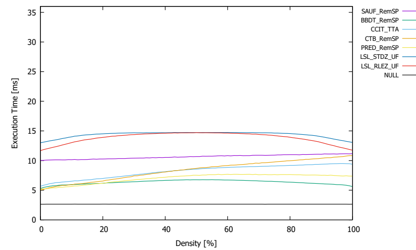
(a) $g = 3$



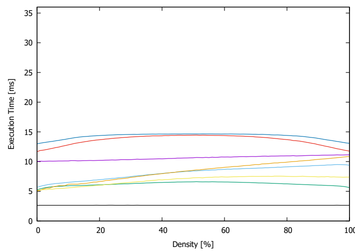
(b) $g = 4$



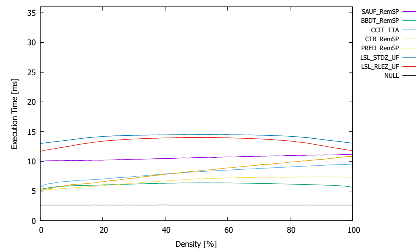
(c) $g = 8$



(d) $g = 10$



(e) $g = 12$



(f) $g = 16$

Figure B.2: Granularity results in ms on an Intel Core i7-4770 *CPU* @ 3.40GHz running Linux with GCC 5.1.0. Lower is better.

A New Paradigm for Sequential CCL Algorithms, 5

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.3) and GCC 5.1.0 (Table B.4) compilers and Linux 4.18.0 (64 bit) OS with GCC 5.5.0 compiler (Table B.5).

Table B.3: 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

This comparison highlights how the performance of an algorithm coupled with a label solver may significantly change with the environment. The RemSP label solver is not always the best choice for Spaghetti. On this specific architecture, the algorithm performs better with UF solver on Windows (both with MSVC and GCC) and with RemSP solver under Linux (with GCC). Additionally, in Table B.6 the same strategy of Spaghetti is also applied to the mask presented in [58], which

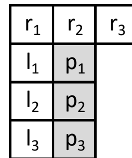


Figure B.3: Scan mask for CTBE [58].

Table B.4: 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

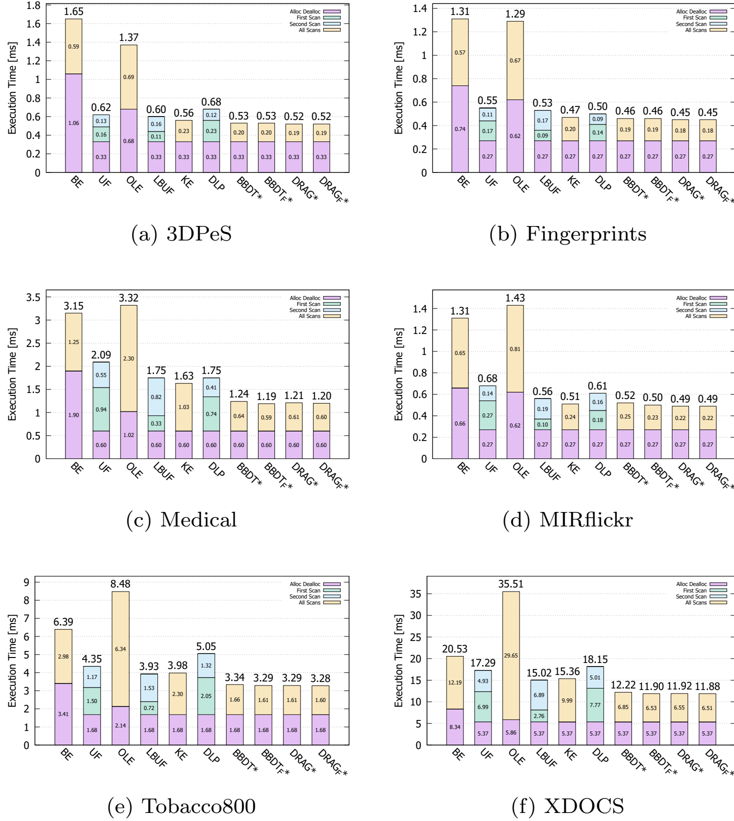
is identified by CTBE (CTB Enhanced to work with three lines). Applying the proposed strategy to this mask (Fig. B.3) lowers the performance, because it is unable to consider the fact that all pixels in a single block share the same label, running the mask twice for blocks which are instead a single 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 [58].

Table B.5: 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.6: Average run-time tests for our algorithm applied to the mask described in [58]. Settings of Table B.3.

	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.4: 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.