Connected Components Labeling on DRAGs: Implementation and Reproducibility Notes

Federico Bolelli, Michele Cancilla, Lorenzo Baraldi, and Costantino Grana

Dipartimento di Ingegneria "Enzo Ferrari"
Università degli Studi di Modena e Reggio Emilia
Via Vivarelli 10, Modena MO 41125, Italy
{name.surname}@unimore.it

Abstract. In this paper we describe the algorithmic implementation details of "Connected Components Labeling on DRAGs" (Directed Rooted Acyclic Graphs), studying the influence of parameters on the results. Moreover, a detailed description of how to install, setup and use YACCLAB (Yet Another Connected Components LAbeling Benchmark) to test DRAG is provided.

1 Introduction

Connected Components Labeling (CCL) is one of the fundamental operations in Computer Vision and Image Processing. With the labeling procedure, all objects in a binary image are labeled with unique values, typically integer numbers. In the last few decades many novel proposals for CCL appeared, and almost none of them was compared on the same data or with the same implementation, introducing reproducibility and evaluation issues. Starting from this discrepancy, the benchmarking framework Yet Another Connected Components LAbeling Benchmark (YACCLAB in short) has been developed, aiming to provide the fairest possible evaluation of CCL algorithms [4,5].

The performance evaluation task is not as easy as it may seem, as there are several aspects that could influence an algorithm. However, since CCL is a well-defined problem, admitting a unique solution, the key elements influencing the "speed" of an algorithm can be reduced to: (i) the data on which tests are performed, (ii) the quality of the implementations, (iii) the hardware capabilities, and last but not least, (iv) the code optimization provided by the compiler.

For these reasons, the YACCLAB benchmark is based on two fundamental traits which aim at guaranteeing the reproducibility of the claims made by research papers:

(i) A public dataset of binary images that covers different application scenarios, ranging from text analysis to video surveillance.
(ii) A set of open-source `C++` algorithms implementations, on which anyone can contribute to, with extensions or improvements.

The results obtained with YACCLAB may vary when the computer architecture or the compiler change, but being the code publicly available, anyone can test the provided algorithms on his own setting, verifying any claim found in literature and choosing the one which suits his needs best.

Following this line of work, in this paper we describe the algorithmic and implementation details of a recently developed CCL algorithm, "Connected Component Labeling on DRAGs" (Directed Rooted Acyclic Graphs) [3], focusing on its integration with YACCLAB and on the installation procedure. A detailed analysis of parameters influence on the result is also provided.

The source code of the aforementioned algorithm is located at [1], whereas all the benchmarking suite can be found at [2].

2 How to test DRAG with YACCLAB

To correctly install and run YACCLAB the following packages, libraries and utilities are required:

- CMake 3.0.0 or higher (`https://cmake.org`);
- OpenCV 3.0 or higher (`http://opencv.org`),
- Gnuplot (`http://www.gnuplot.info`);
- a C++11 compiler.

The installation procedure requires the following steps:

- Clone the GitHub repository [2];
- Install the software using CMake, which should automatically find OpenCV path, whether correctly installed on your OS, download the YACCLAB dataset, and create a `C++` project for the selected compiler;
- Set the configuration file `config.yaml` placed in the installation folder and select the desired tests;
- Open the project, compile and run it.

There are six different tests available: *correctness* tests are an initial validation of the algorithms with border cases; *average* tests run algorithms on every image of a dataset, reporting for each method the average run-time; *average_with_steps* separates the labeling time of each scan, and that required to allocate/deallocate data structures; *density* and *granularity* use synthetic images to evaluate the performance of different approaches in terms of scalability on the number of pixels, foreground density and pattern granularity; *memory* tests report an indication on the expected number of memory accesses required by an algorithm on a reference dataset.

YACCLAB stores average results in three different formats: a plain text file, histogram charts, either in color or in gray-scale, and a LaTeX table, which can be directly included in research papers. If an algorithm employs multiple scans, results will display time spent in each of them separately, producing a stacked histogram chart as output.

All the algorithms included in YACCLAB employ a base interface and implement the following virtual methods:

- `PerformLabeling`: includes the whole algorithm code and it is necessary to perform average, density, granularity and size 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 Union-Find strategy is independent from the CCL one, therefore all CCL algorithms invoke a templated Union-Find implementation. YACCLAB is then 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), Interleaved Rem's algorithm with splicing (RemSP) and Three Table Array (TTA). This standardization reduces the code variability, allowing to separate the Union-Find data structures from the ones of CCL algorithms, and provides fairer comparisons without negatively impacting the execution time.

The *NULL labeling*, also referred as NULL, defines a lower bound limit for the execution time of CCL algorithms on a given machine and a reference dataset. As the name suggests, the *NULL labeling* does not provide the correct connected components of an image, but only copies the pixels from the input image into the output one. This "algorithm" allows to identify the minimum time required for allocating the memory of the output image, reading the input image and writing the output one. In this way, all the algorithms can be compared in terms of how costly the additional operations required are.

3 Experiments Reproducibility

The DRAG algorithm was tested 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 the `-O3` and `-m32` flags.

The impact of the labels solver on the overall performance is typically limited for most algorithms, so we only reported results obtained with the UFPC solver on the state-of-the-art algorithms.

The DRAG performance have been compared on six different datasets: a collection of histological images with an average amount of 1.21 million pixels to analyze and 484 components to label (Medical), fingerprint images collected by using low-cost optical sensors or synthetically generated with an average of 809 components to label (Fingerprints), high resolution historical document images with more than 15000 components and a low foreground density (XDOCS), a dataset for people detection, tracking, action analysis and trajectory analysis with very low foreground density and few components to identify (3DPeS), a selection of documents collected and scanned using a wide variety of equipment over time with a resolution varying from 150 to 300 DPI (Tobacco800), and a large set of standard resolution natural images taken from Flickr (MirFlickr).

In order to execute the same experiments reported in [3] the `perform`, `algorithms`, and `average_datasets` fields in the configuration file must be set as follows:

```
average_datasets: ["mirflickr", "fingerprints", "xdocs", "tobacco800",
                   "3dpes", "medical"]
algorithms: [SAUF_UFPC, BBDT_UFPC, DRAG_UFPC, CTB_UFPC, PRED_UFPC, CT,
             labeling_NULL]
perform: {correctness: true, average: true, average_with_steps: false,
          density: false, granularity: false, memory: false}
```

Average tests were repeated 10 times (setting the `tests_number.average` in the configuration file), and for each image the minimum execution time was 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 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.

4 Conclusion

This paper describes how to setup the YACCLAB project to reproduce the result reported in [3]. The processor model –and in particular the cache sizes–, the RAM speed and the background tasks will influence the execution time. Nevertheless, the algorithms relative performance should remain extremely similar. Changing the OS or the compiler is instead likely to heavily influence the outcome.

References

1. The DRAG Algorithm. https://github.com/prittt/YACCLAB/blob/master/include/labeling_bolelli_2018.h, accessed on 30-July-2018
2. The YACCLAB Project. https://github.com/prittt/YACCLAB, accessed on 30-July-2018
3. Bolelli, F., Baraldi, L., Cancilla, M., Grana, C.: Connected Components Labeling on DRAGs. In: 24rd International Conference on Pattern Recognition (ICPR) (Aug 2018)
4. Bolelli, F., Cancilla, M., Baraldi, L., Grana, C.: Toward reliable experiments on the performance of Connected Components Labeling algorithms. Journal of Real-Time Image Processing pp. 1–16 (2018)
5. Grana, C., Bolelli, F., Baraldi, L., Vezzani, R.: YACCLAB - Yet Another Connected Components Labeling Benchmark. In: 23rd International Conference on Pattern Recognition (ICPR). pp. 3109–3114 (Dec 2016)