Linear Variation and Optimization of Algorithms for Connected Components Labeling in Binary Images

Fedor Alekseev[1], Mikhail Alekseev[23], and Artyom Makovetskii[24]

[1] Moscow Institute of Physics and Technology State University, Dolgoprudny, Russia
alekseev@phystech.edu
[2] Chelyabinsk State University, Chelyabinsk, Russia
[3] alexeev@csu.ru
[4] artemmac@mail.ru

Abstract. Linear variation is a topological characteristic of a function of two variables. The problem of linear variation computing can be reduced to the problem of counting connected components in a binary image with eight-connected connectivity. The proposed method is essentially a modification of some known raster algorithms for connected components labeling that groups pixels into 2×2 cells. A performance benchmark of the proposed method applied to some approaches on random images is provided.

Keywords: Binary raster image, connected component labeling, pattern recognition

1 Introduction

Continuous functions of two variables have a set of topological characteristics referred to as linear variations. Kronrod[1] introduced the notion of regular component of the level set of continuous function of two variables. The simplest topological characteristic in linear variation theory is the number of regular components for all level sets of a function. Works [2,3,4] address the necessity of linear variation in image restoration theory.

The problem of computing linear variation in discrete case is equivalent to the problem of counting the number of connected components (CC) in a binary image with 8-connectivity. This is usually done through CC labeling, and many approaches for that are known[5]. We came to the labeling problem while searching an efficient algorithm algorithm for linear variation computation. Of course, the labeling problem also is well known in the digital image processing theory.

This paper presents an efficient way to convert the initial binary image with 8-connectivity into a condensed (non-binary) image with new connectivity that is 2 times smaller in each dimension than the initial image. Most approaches for CC labeling are still valid for the new image. As time and memory consumptions of conventional approaches usually depend linearly on the number of pixels,

running them on the condensed image can be more efficient than running on the initial image.

It is worth noting that our method exploits an important property of 8-connectivity that does not hold for 4-connectivity, so it is not applicable in the latter case.

2 2×2 condensation

We will use the fact that in case of 8-connectivity all 4 pixels of any 2×2 square are adjacent to each other, and so are contained in same CC and should have same labels upon completion of the labeling algorithm. So instead of labeling single pixels we can label 2×2 groups of pixels, or "cells".

Consider a $2N \times 2M$ binary image[5] specified by binary predicate $b(x, y)$, returning 1 in case the pixel at the given coordinates is an object pixel, and 0 otherwise (background pixel). For convenience we use 0-indexation throughout the paper, so the topmost row is associated with row index $y = 0$, and the leftmost column is associated with column index $x = 0$.

The condensed image will be of size $N \times M$. Each pixel of the condensed image will unambiguously represent the configuration of the corresponding 2×2 cell of pixels of the initial binary image. So the condensed image will contain pixels of $2^4 = 16$ colors. We suggest the function $c(x, y)$ denoting the color of a pixel of condensed image to be the following:

$$
\begin{aligned}
c(x, y) = {} & 2^0 b(2x, 2y) + 2^1 b(2x + 1, 2y) \\
& + 2^2 b(2x, 2y + 1) + 2^3 b(2x + 1, 2y + 1)
\end{aligned}
$$

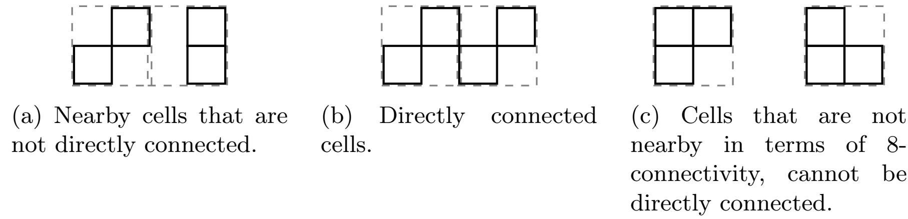
So the value of each pixel of the initial image is stored in the corresponding bit in color of the corresponding pixel of the condensed image. Note that if $c(x, y) = 0$ for some (x, y), then there are no object pixels in this cell, and no label should be assigned to this whole cell. The enumeration of pixels inside one cell is shown on Figure 1.

0	1
2	3

Fig. 1: Cell pixels enumeration

We need also to define a connectivity function for the condensed image. We cannot actually just use 8-connectivity, as whether two cells are adjacent cells of one CC depends not only on their coordinates, but also on their configuration. See Figure 2 for examples.

[5] For convenience in this paper we consider only input images with even size in both dimensions. If this is not the case, one can easily append a row or a column of background pixels to the image.



(a) Nearby cells that are not directly connected.

(b) Directly connected cells.

(c) Cells that are not nearby in terms of 8-connectivity, cannot be directly connected.

Fig. 2: Cells connectivity examples

.

However, for every possible case of relative placement of two nearby cells there is a mask of pixels of each cells that are important for direct connectivity of this pair of cells. So two cells are considered directly connected, if they are nearby in terms of 8-connectivity and each of them contains at least one object pixel in the mask of important pixels corresponding to relative placement of the cells. The masks are shown in Figure 3.

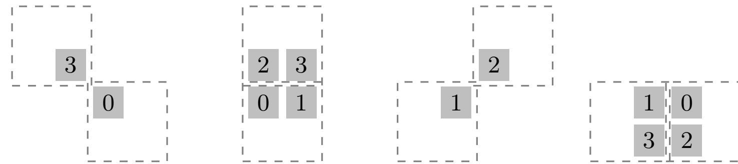


Fig. 3: Masks of important pixels for all four cases of relative placement of two nearby cells. Those pixels are filled in gray, with their numbers within cell specified.

```
if (b(x, y)) {
  if (b(x, y-1)) {
    // copy labels, or union label sets
  }
  // consider other directions
}
```

(a) Binary image with 8-connectivity.

```
if (c(x, y)) {
  if ((c(x, y-1) & ((1<<2) | (1<<3)))
   && (c(x, y)   & ((1<<0) | (1<<1)))) {
    // copy labels, or union label sets
  }
  // consider other directions
}
```

(b) Condensed image with new connectivity.

Fig. 4: Checking the nearby top pixel/cell.

The idea is that if we naturally encode mask pixels as 4-bit numbers in the way similar to how we defined c, we can express the predicate denoting whether two nearby cells are directly connected efficiently with just two bitwise and one logical AND operations. As an example, Figure 4 compares possible codes in `C`

checking if the labels of the current pixel or cell and the one right on top of it should be same for classic 8-connectivity and for the new connectivity. Note that as bitwise operations are usually relatively cheap, we added only a little complication compared to the decrease of the number of pixels to process by the factor of 4.

3 Benchmarks

In this section we present performance benchmarks of some popular methods for counting CC both with and without the 2×2 cells optimization.

3.1 Algorithms overview

As each of these methods, omitting DTSUF could be used with both condensed and non-condensed images, for convenience we denote height and width of the image on input of each algorithm as H and W.

Also to avoid repeating "pixel or cell" we will just use the word "pixel" meaning the appropriate raster for non-condensed and condensed images. When mentioning neighbors of some pixel, we mean pixels that are directly connected to it in terms of appropriate connectivity.

DFS The first algorithm we chose for the benchmark is Depth First Search graph traversal. Graph traversal algorithms are the general approach for CC labeling in graphs, and DFS is arguably the most simple. Although not intended for image labeling and suboptimal in most cases, it still has linear in the number of pixels worst-case complexity in both time and memory.

Like most approaches, the algorithm has an outer loop over all pixels of the image. Each time an object pixel (non-empty cell, in case it is actually a cell) which is not labeled yet is encountered, DFS traversal is launched from that pixel. The traversal labels all the object pixels of the same CC, launching recursively from all unlabeled neighbors of current pixel.

SUF Another general approach for graph CC labeling which is more popular in image labeling is using some kind of a data structure to maintain disjoint sets of pixels that are considered to be in the same CC.

The structure is commonly called Union-Find, as the two essential operations it is supposed to support are Union, which is used to merge some two of the disjoint sets in one, and Find, which is used to find the representative element of the set some element belongs to, so that we can always tell if two elements are in the same set. The structure can be implemented so that the amortized time complexity for each operation is $O(\alpha(HW))$, where α is the inverse Ackermann function, and $\alpha(HW) < 5$ for all remotely practical image sizes[6].

The approach that is named SUF (Scan with Union-Find) in this paper follows.

Pixels are processed in the natural order: from top to bottom and from left to right, so each row is fully processed before all rows after it. So for each object pixel we encounter there are at most 4 nearby pixels that were already processed, see Figure 5.

At first each object pixel is considered to belong to its own CC, so right before considering the neighbors we increment the CC counter. Moreover, it is associated with its own disjoint set in the Union-Find structure. More specifically, object pixel at position x, y is associated with the set number $xW + y$ to guarantee absence of collisions.

The possible neighbors are processed in the order shown on Figure 5, and for each neighbor that is object pixel, the Union operation is performed on the two sets containing the current and the neighbor pixel. In case the Union operation was successful, which means that the pixels had belonged previously to different sets and we have just found a way to merge two CC, the CC counter is decremented.

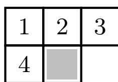


Fig. 5: Neighbor mask for SUF algorithm family.

Note that this algorithm outputs not the matrix of final labels, but only the number of CC. This is the place in which it differs from some common two-pass approaches for CC labeling, as it is an algorithm not for labeling, but just for counting CC. The latter is exactly what is needed for the aforementioned linear variation problem.

Also we have to mention that for performance reasons the Union-Find internal structures are allocated ahead to have the size of at least HW to guarantee that each object can be easily associated with unique starting set given only its coordinates. So memory consumption of SUF is linear in the number of pixels ($O(HW)$), while its amortized time complexity is "almost" linear ($O(HW\alpha(HW))$).

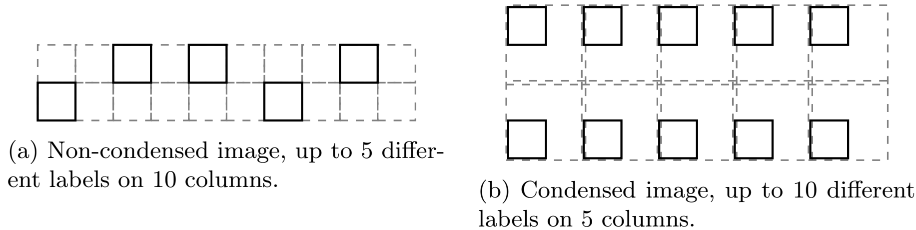
SUF2 This approach is a modification of SUF designed to reduce memory consumption and potential number of cache-misses generated by Union-Find operations.

The idea is to run through each row twice: first time to establish label equivalence classes within row, and the second time to set unique equal label for all pixels of same CC. This gives us the opportunity to maintain just $O(W)$ different labels at each moment, and so the Union-Find structure can be allocated for only $O(W)$ elements, as opposed to WH elements needed for SUF.

The exact maximum number of label equivalence classes between two adjacent rows depends on whether the given image is condensed. For non-condensed

images $W = 2M$, and maximum number of label equivalence classes in two adjacent rows is $M = \frac{W}{2}$, as if two object pixels are both in adjacent rows and in adjacent columns, then they are neighbors in terms of 8-connectivity.

On the other hand, for condensed images $W = M$, and maximum number of label equivalence class in two adjacent rows of cells is $2W = 2M$. See Figure 6 for illustration on the extremal cases.



(a) Non-condensed image, up to 5 different labels on 10 columns.

(b) Condensed image, up to 10 different labels on 5 columns.

Fig. 6: Extremal cases for SUF2.

Our benchmarks show that, although SUF2 without condensation is generally more efficient on noise images than SUF without condensation, the mentioned consideration make the 2×2 optimization of SUF2 less dramatic in means of performance increase factor. Optimized SUF2 significantly outperforms optimized SUF only on huge noise images with less than 60% density, while falling behind on small and medium-sized noise images.

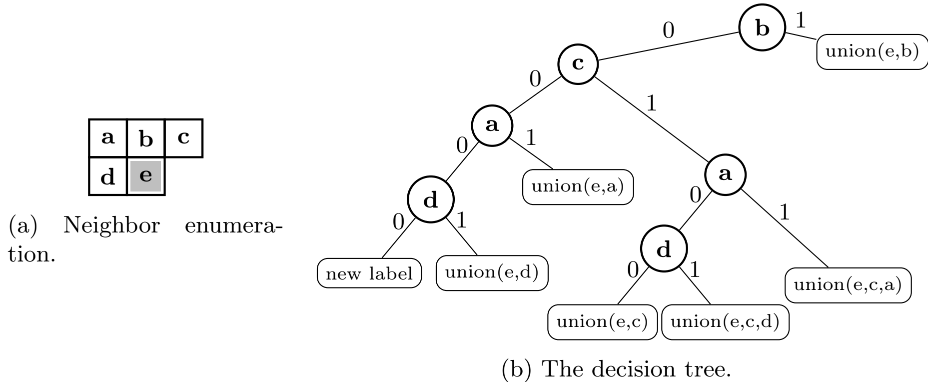
Note that, while SUF2 is intended for CC counting, its memory consumption is only $O(W)$, which is significant for embedded systems and other resource-tight applications. The time complexity is same as that of SUF, i. e. $O(HW\alpha(HW))$.

DTSUF This modification of SUF was implemented to compare the optimization we present with another optimization suggested by Wu, Otoo and Suzuki[7], namely decision tree. Unfortunately DTSUF (Decision Tree Scan plus Union-Find) is not applicable for condensed image, as it exploits similar observations on 8-connectivity. So this approach is the direct competitor to our optimizations of SUF.

As before, the goal is to reduce the number of pixel lookup and label copy operations. This is achieved with careful consideration of only the necessary neighbors, stopping once the set of one or two label operations that are needed is determined. The decision tree for processing an object pixel is shown on Figure 7.

3.2 Condensation types

For memory-tight applications such as embedded systems it can be useful to utilise the suggested condensation in a lazy way, so that without actually constructing the condensed image we use a function ("view") calculating color of



(a) Neighbor enumeration.

(b) The decision tree.

Fig. 7: DTSUF decision tree optimization.

each cell with 4 lookups to the original binary image. This is probably most interesting when combined with SUF2 algorithm, as it is a way to win some time while avoiding memory footprint increase. However this lazy approach increases the number of image lookups.

So the benchmark charts show three levels of optimization applied to DFS, SUF and SUF2: just method name in legend refers to method applied to non-condensed image, " on 2x2 view" means lazy memory-less condensation and " on 2x2 copy" means that a temporary condensed image was constructed in memory. For the latter case, the construction time was included into the overall measured time of run.

3.3 Measurement

For measurement purposes all algorithms and optimizations were implemented by us in `C++` .

The program was compiled using `g++ 5.3.0-3` compiler with `-O3` optimization flag. All tests were run on same machine running under Arch Linux with standard `linux 4.4.1-2` kernel, utilising non-overclocked `Intel(R) Core(TM) i5-2430M CPU @ 2.40GHz` (L3 cache size 3072K) processor and non-overclocked 1333MHz 8GiB Kingston RAM.

The test images were random images, generated in the following manner. At first, for given density d, the first $4dNM$ pixels of the $2N \times 2M$ image were filled with object values, while rest filled with background values. Then all the pixels were permuted using `std::shuffle` function from `C++` standard library, which is an implementation of Fisher-Yates Shuffle algorithm.

All methods were run on same tests, which was achieved by supplying same seed to the `std::mt19937` Mersenne Twister[8] random number engine implementation from `C++` standard library before passing the engine to `std::shuffle`.

Benchmarks showed that for random images, the run time depends non-trivially on the value of density d, which is the ratio of object pixels in the image.

On the one hand, this is because all of the considered algorithms pass background pixels without any processing, and somehow process the object pixels, so the exact number of operations such as Union and Find should generally grow with d. On the other hand, the greater the value of d, the less number of CC is likely to be present in the image, which reduces the number of relatively expensive Union operations.

So benchmarks were run for all $d \in \{ 2\%, 5\%, 10\%, 15\%, \dots, 90\%, 95\%, 98\% \}$ to achieve a picture of how the speedup depends on the number of object pixels in the image.

It is worth noting again that for "2x2 copy" optimization, the apparent time overhead of constructing the extra condensed image was included into the total run time, since we measured the total run time of the "black box" CC counting subroutine, supplied with only the original image, so that all the needed preparations are done inside the subroutine.

3.4 Results

The overall benchmark plot of average run time for all algorithms and optimizations is presented on Figure 8. The actual measured points are known for sure only for the mentioned values of d, the points are connected to help distinguish groups of points corresponding to same algorithms.

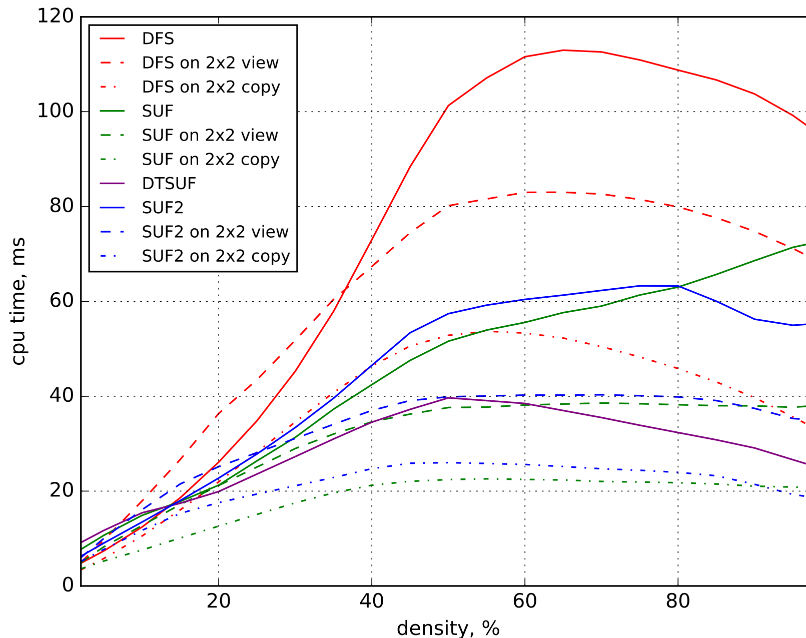


Fig. 8: Average CPU time per run on random images 1200×1800 px.

To show more closely how efficient the optimizations actually appear to be, we plotted speedup/density charts for the considered methods (Figure 9). Here the speedup of algorithm A relative to algorithm B is defined as $\frac{T_B}{T_A}$, where T_A and T_B are the total times used by the corresponding algorithms to count CC in all the test images of same density. The greater the speedup of optimized method relative to original method, the more efficient the optimization is considered.

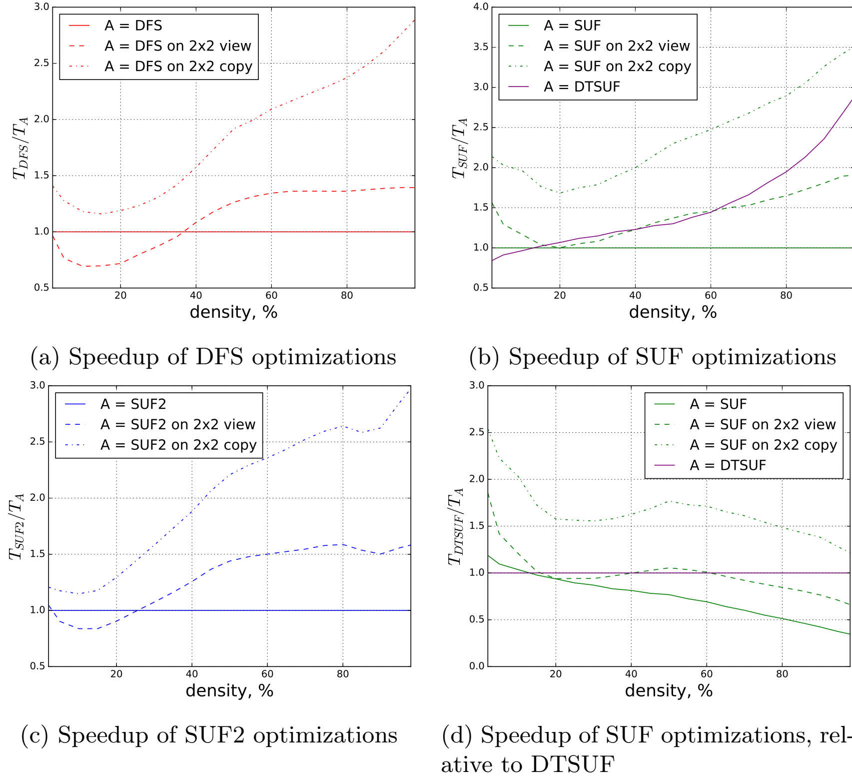


Fig. 9: Speedup/density charts

The speedup plots on Figures 9a to 9c suggest that the "2x2 copy" optimization speeds up the considered methods by factor greater than 1 for all tested densities. On the other hand, the "2x2 view" optimization is showed to be not time-efficient at all for some densities, slowing down DFS and SUF2 if run on images with densities less than 35% and 25% respectively. Utilizing the latter optimization also does not speed up SUF on images on density approaching 20%, although it does not slow down the algorithms either.

The worst-case and best-case speedups are shown on the Table 1. Here we see that while "2x2 view" optimization is more memory-efficient than "2x2 copy",

it can slow down the original method in some situations, and generally it does not give as much performance advantage.

We should also mention the chart on Figure 9d, featuring speedup of our SUF optimizations relative to DTSUF in order to compare our optimizations to the optimization by Wu, Otoo and Suzuki. The plot clearly shows that for all the densities under 80% our "SUF on 2x2 copy" algorithm outperforms the decision-tree based approach by the factor of at least 1.5, with speedup reaching 2.5 for 2% density.

Algorithm	worst speedup	worst speedup density	best speedup	best speedup density
DFS on 2x2 view	0.7	10	1.4	65
DFS on 2x2 copy	1.2	10	2.9	98
SUF on 2x2 view	1.0	15	1.9	95
SUF on 2x2 copy	1.7	20	3.5	98
DTSUF	0.8	2	2.9	98
SUF2 on 2x2 view	0.8	10	1.6	75
SUF2 on 2x2 copy	1.1	10	3.0	98

Table 1: The worst and the best speedups relative to original methods. The shown speedup of DTSUF is relative to SUF.

4 Conclusion

We present an efficient way to convert the initial binary image with 8-connectivity into a smaller condensed image with new connectivity. Most approaches for connected components labeling are still valid for the new image. As time and memory complexities of conventional approaches usually depend linearly on the number of pixels, running them on the condensed image can be more efficient than running on the initial image. The provided benchmark on random images showed worst-case speedup factor of 1.7 for one of our optimizations of the common SUF approach to CC labeling. The same algorithm is shown to outperform the previously known decision tree optimization of SUF by at least 1.5 factor for images with less than 80% density.

5 Acknowledgements

The work was supported by the Ministry of Education and Science of Russian Federation (grant 2/1766/2014K).

References

1. Kronrod, A.: On functions of two variables. Uspehi Mat. Nauk 5, 1(35), 24-134(1950)

2. Makovetskii, A., Kober, V.: Image restoration based on topological properties of functions of two variables. Proc. SPIE Applications of Digital Image Processing XXXV, 8499, 84990A (2012)
3. Makovetskii, A., Kober, V.: Modified gradient descent method for image restoration. Proc. SPIE Applications of Digital Image Processing XXXVI, 8856, 885608-1 (2013)
4. Chochia, P., Milukova, O.: Comparison of Two-Dimensional Variations in the Context of the Digital Image Complexity Assessment. Journal of Communications Technology and Electronics, 2015, 60, no. 12, pp. 1432-1440
5. He, L., Chao, Y., Suzuki, K., Wu, K.: Fast connected-component labeling. Pattern Recognition. v42, Pages 1977-1987. (2009)
6. Cormen, H., Leiserson, E., Rivest, L., Stein, C.: Introduction to Algorithms (Second ed.). MIT Press (2001)
7. Wu, K, Otoo, E., Suzuki, K.: Optimizing two-pass connected-component labeling algorithms. Pattern Anal. Appl. (2008)
8. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation 8 (1): 3–30. (1998)