Improving the Performance of Thinning Algorithms with Directed Rooted Acyclic Graphs

Federico Bolelli 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 propose a strategy to optimize the performance of thinning algorithms. This solution is obtained by combining three proven strategies for binary images neighborhood exploration, namely modeling the problem with an optimal decision tree, reusing pixels from the previous step of the algorithm, and reducing the code footprint by means of Directed Rooted Acyclic Graphs. A complete and open-source benchmarking suite is also provided. Experimental results confirm that the proposed algorithms clearly outperform classical implementations.

Keywords: Thinning · Skeletonization · Optimization · Decision Trees · Binary Image Processing.

1 Introduction

Thinning is a 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 [10]. Firstly introduced in the 1950 as a data compression strategy [11], the thinning procedure is nowadays used as a pre- or post-processing step in many different applications, ranging from medical imaging [33,34] to handwritten text recognition [7,19] and fingerprint analysis [22]. 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 [35] is one of the most famous and used, given its efficiency and simplicity. This algorithm is based on the 8-neighbor connectivity and exploits two sub-iterations that are iteratively performed to remove pixels and obtain the final result. In [8], 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.* [20] tackled the problem by a different perspective and proposed an improvement on the Zangh-Suen technique which requires less iterations, at

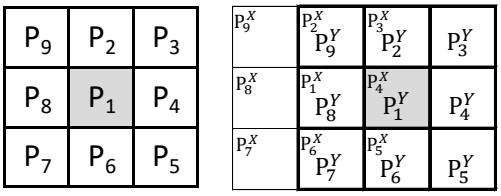


Fig. 1. Naming convention for the pixel in the neighborhood of P_1 (a) and their overlap when the mask is shifted for processing the next pixel (b).

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 [16].

The algorithm by Guo and Hall [15] 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 [25].

These solutions have been proposed some decades ago, but are still commonly used [22,33,34] and included in many image processing libraries, such as OpenCV [27].

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 [13]. 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.

Both of these approaches miss a classical optimization strategy used when working with local neighborhoods: when the scanning mask moves horizontally most of the pixels have already been read in the previous step (Fig. 1), so only the rightmost column needs to be read, and the others can be obtained by just shifting the positions of the previously inspected ones. This optimization approach is typically used with average/box filtering, running median [21], and Connected Components Labeling [18]. A solution for combining this *prediction* with DTrees was introduced in [12].

Moreover, in [3,6], a novel approach to model decision problems as Directed Rooted Acyclic Graphs (DRAGs) was introduced. Differently from DTrees, in which the same set of conditions required to reach the corresponding leaf may be checked in multiple subtrees, in a DRAG, being it a graph, these could be merged together. Even though this approach does not save any condition check with respect to the use of a DTree, it allows to sensibly reduce the number of

Algorithm 1 Two subiteration thinning algorithm

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

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

machine instructions, and thus the impact on instruction cache. Indeed, the code generated from a DRAG will include the same checks only once.

With this paper we extend the DRAG model in order to apply it on state-of-the-art thinning algorithms and improve their performance. Moreover, we apply a solution to include a prediction strategy with DRAGs. To evaluate the effectiveness of our proposals and compare them with existing implementations, an open-source *C++* benchmarking system has also been developed. The source code of the benchmark, as well as the proposed algorithms, is available in [31].

2 Thinning Algorithms

Many thinning algorithms belong to the class of parallel thinning algorithms [23]: 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, which work with biased subiterations: 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 1).

The notation used in the algorithms is summarized here. Given I, an image defined over a two dimensional rectangular lattice $\mathcal{L}$, and $I(p)$ the value of pixel $p \in \mathcal{L}$, with $p = (p_x, p_y)$, we define the *neighborhood* of a pixel as follows:

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

Two pixels, p and q, are said to be *neighbors* if $q \in \mathcal{N}(p)$, that implies $p \in \mathcal{N}(q)$. From a visual perspective, p and q are *neighbors* if they share an edge

Algorithm 2 Removal logic functions for ZhangSuen and ChenHsu algorithms

function A(P)
    return $(\neg P_2 \wedge P_3) + (\neg P_3 \wedge P_4) + (\neg P_4 \wedge P_5) + (\neg P_5 \wedge P_6) +$
        $(\neg P_6 \wedge P_7) + (\neg P_7 \wedge P_8) + (\neg P_8 \wedge P_9) + (\neg P_9 \wedge P_2)$
function B(P)
    return $P_2 + P_3 + P_4 + P_5 + P_6 + P_7 + P_8 + P_9$

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

function CH_SHOULD_REMOVE(I, p, k)
    $P \leftarrow I(\mathcal{N}(p))$
    if $k = 0$ **then**
        $c \leftarrow P_2 \wedge P_4 \wedge P_6$;
        $d \leftarrow P_4 \wedge P_6 \wedge P_8$;
        $f \leftarrow P_2 \wedge P_4 \wedge \neg P_6 \neg P_7 \neg P_8$
        $g \leftarrow P_4 \wedge P_6 \wedge \neg P_2 \neg P_8 \neg P_9$
    else
        $c \leftarrow P_2 \wedge P_4 \wedge P_8$;
        $d \leftarrow P_2 \wedge P_6 \wedge P_8$;
        $f \leftarrow P_2 \wedge P_8 \wedge \neg P_4 \neg P_5 \neg P_6$
        $g \leftarrow P_6 \wedge P_8 \wedge \neg P_2 \neg P_3 \neg P_4$
    return $(2 \leq \mathrm{B}(P) \leq 7) \wedge ((\mathrm{A}(P) = 1) \wedge \neg c \wedge \neg d) \vee$
        $(\mathrm{A}(P) = 2) \wedge (f \vee g))$

or a vertex. The set defined in Eq. 1 is called 8-neighborhood of p. 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. The conditions for foreground pixel removal depend on the neighborhood and the algorithm flavor. Following the original notation of Zhang and Suen, pixels are enumerated in clockwise order, with the current pixel being P_1.

Algorithm 2 and Algorithm 3 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. The basic idea is to remove pixels at foreground

Algorithm 3 Removal logic function for GuoHall algorithm

function GH_SHOULD_REMOVE(I, p, k)
  $P \leftarrow I(\mathcal{N}(p))$
  $C \leftarrow ((\neg P_2) \wedge (P_3 \vee P_4)) + ((\neg P_4) \wedge (P_5 \vee P_6)) +$
    $((\neg P_6) \wedge (P_7 \vee P_8)) + ((\neg P_8) \wedge (P_9 \vee P_2))$
  $N1 \leftarrow (P_9 \vee P_2) + (P_3 \vee P_4) + (P_5 \vee P_6) + (P_7 \vee P_8)$
  $N2 \leftarrow (P_2 \vee P_3) + (P_4 \vee P_5) + (P_6 \vee P_7) + (P_8 \vee P_9)$
  $N \leftarrow min(N1, N2)$
  if $k = 0$ **then**
    $m \leftarrow (P_6 \vee P_7 \vee \neg P_9) \wedge P_8$
  else
    $m \leftarrow (P_2 \vee P_3 \vee \neg P_5) \wedge P_4$
  return $(C = 1) \wedge (2 \leq N \leq 3) \wedge \neg m$

connected components edges (*i.e.*, the block should not be totally foreground), without splitting that component.

Chen and Hsu [9] observed that given the eight neighbors of P_1 the outcome of the conditions is known, thus they built two lookup tables (LUT) 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.

3 Techniques for Performance Optimization

The LUT approach suggests that thinning techniques can be modeled as decision tables [29]. A decision table is a tabular form that presents a set of conditions and their corresponding actions. A statement section reports a set of conditions which must be tested and a list of actions to perform. Each combination of condition entries (*condition outcomes*) is paired to an *action entry*. In the action entries, a column is marked to specify whether the corresponding action is to be performed or not. The aforementioned thinning algorithms can thus be modeled as a decision table in which the conditions are given by the current pixel and its neighborhood, and the only two possible actions are removing the current pixel or not. By plugging the subiteration index k as another condition, this results in a 10 conditions decision table (1024 rules).

The definition of decision tables requires all conditions to be tested in order to select the corresponding actions to be executed. Testing the conditions requires to access the corresponding pixel in the image, so solutions to avoid checking conditions allow to improve the algorithm computational requirements.

With a dynamic programming technique, Grana *et al.* [14] showed how to build an optimal decision tree that, by saving many memory accesses, resulted in a considerable improvement in execution times.

Following [12], we observe that it is possible to include *prediction* in the DTree, by keeping track of the examined pixels on the path to reach a leaf, and

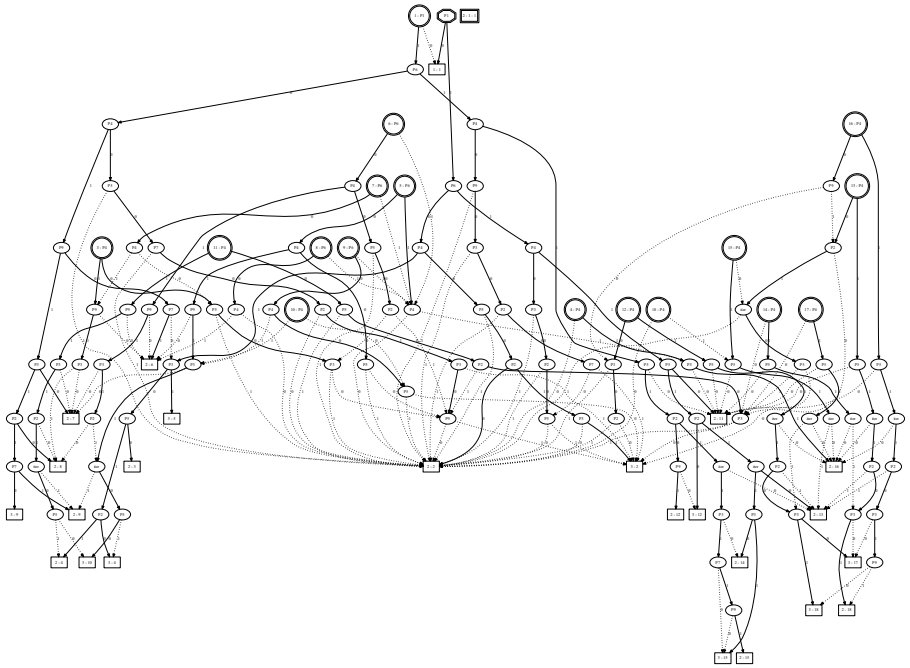


Fig. 2. Final DRAG for Zhang and Suen thinning algorithm. The octagonal shaped root is the starting node for the first pixel in a line, double circles are roots from which the algorithm will restart after reaching a leaf (the first number is their id), ellipses are decisions and rectangles are leaves. 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 on the online version.

there selecting a different tree which employs only the unknown pixels. While this requires a pretty tedious work, the number of possible combinations is quite limited and it is doable. For instance, for the ZS algorithm, we obtain 18 different reduced trees. Its possible to note that P_8 disappears from all trees, because it is always known from the previous step (it was P_1). Moreover a degenerate case happens, because if P_4 was a background pixel, the next trees will have a root which directly contains the action "do nothing".

Additionally, as introduced in [3], a transformation from a DTree to a DRAG can be performed by substituting all equal subtrees with a single instance by making every parent node point to that unique exemplar. We can traverse the tree and, for every subtree, search an equal one and immediately perform the substitution. This transformation does not depend on the order in which the original tree is traversed. 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. A visualization of the resulting graph for the Zhang and Suen algorithm is shown in Fig. 2.

4 Comparative Evaluation

The proposed algorithms are evaluated by comparing their performance with state-of-the-art implementations. There are many variables that could influence the performance of an algorithm in terms of execution time: the machine architecture and the operating system on which tests are performed, the adopted compiler and its optimization settings, code implementation and last but not least the data on which algorithms are tested. In order to ensure experiment reproducibility and allow researchers to test and compare the algorithms on their own settings, an open-source benchmarking system called THeBE (the THinning evaluation BEnchmark) has been designed and released. The source code of THeBE and the algorithms implementations are available in [31].

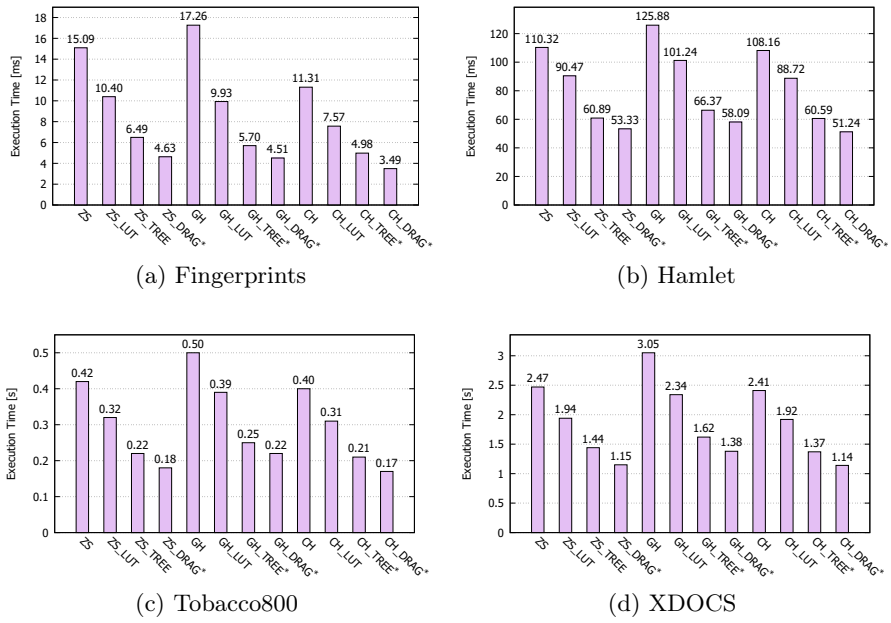
Experimental results reported and discussed in this Section are obtained running THeBE on an Intel Core i7-4790K CPU (with 4×32 KB L1 cache, 4×256 KB L2 cache, and 8 MB of L3 cache), under Windows (64 bit) OS and using the MSVC 19.16.27030.1 compiler with all optimizations enabled.

Tests have been performed on four different datasets that cover most of the scenarios in which the thinning operation is usually applied:

- *Hamlet* is a set of 104 images, scanned from a version of the Hamlet found on the Gutenberg Project [17]. Images have an average amount of 2.71 million of pixels to analyze. This set of images has been already used in a previously published paper to measure the performance of thinning algorithms [14].
- *Tobacco800* is composed of 1290 document images and it is a realistic collection for document image analysis research. These documents were collected and scanned using a wide variety of equipment over time. Images size ranges from 1200×1600 to 2500×3200 pixels [1,24,30].
- *XDOCS* is a collection of high resolution historical document images taken from the large number of civil registries available since the constitution of the Italian state [2,4,5]. XDOCS is composed of 1677 images with an average size of 4853×3387.
- *Fingerprints* counts 960 fingerprint images taken from three fingerprint verification competitions (FCV2000, FCV2002 and FCV2004) [26]. Images were collected by using low-cost optical sensors or synthetically generated. In order to fit them for a thinning application, fingerprints have been binarized using an adaptive threshold [28] and then negated. Resulting images have a size varying from 240×320 up to 640×480 pixels.

All images are provided in 1 bit per pixel PNG format, with 0 being background and 1 being foreground. The aforementioned datasets can be automatically downloaded during the installation of THeBE or they can be found in [32].

The results of the comparison are reported in Fig. 3. For convenience, all the acronyms used in this section are summarized in the following. ZS identifies the



(a) Fingerprints (b) Hamlet

(c) Tobacco800 (d) XDOCS

Fig. 3. Average run-time test on different datasets. Results are obtained under Windows (64 bit) OS with MSVC 19.16.27030.1 using an Intel Core i7-4790K CPU. For the sake of readability numbers are given in ms in (a) and (b), while they are give in s in (c) and (d). Our proposals are identified with *. Lower is better.

Zang and Suen algorithm originally presented in [35], GS is the algorithm by Guo and Hall [15], and CH is the algorithm proposed by Chen and Hsu in [8]. Moreover, the acronym LUT identifies the lookup table implementation of a given algorithm, TREE represents the version of the algorithm based on optimal decision trees (thus without prediction), and DRAG identifies the application of the complete pipeline proposed. It is important to note that both the "standard" and LUT versions of the algorithms also use prediction, avoiding to read pixels that have already been read in the previous step.

All the variations of a given algorithm (standard, LUT, TREE, and DRAG) differ only in execution time, and always produce the same output on the same input image. This is directly verified by THeBE.

Keeping in mind that the three thinning algorithms (ZS, GH, and CH) produce different results, and should be selected based on the task needs, we can observe that CH always shows the best performance. As reported in literature, the use of LUT always improves performance of about 25% with respect to the standard.

Even thought the LUT version of the algorithms employs prediction, the implementation based on optimal decision trees, which does not, performs better (about 50% wrt the standard). This can be explained considering how the LUT

and the TREE version of the algorithms work. The prediction applied on the LUT table is able to avoid the condition check of six pixels at each step of the scanning phase. On the other hand, the TREE version requires to check up to 9 pixels in the worst case but just one pixel in the best one. On average, it is able to avoid the read of more than six pixels at each step of the scanning phase.

The DRAG version of the algorithms, combining the benefit of both prediction and decision trees, always improves for an average total speed-up of about 60% compared to the the standard. The speed-up of DRAG with respect to TREE is 20% for ZS and CH and 15% for GH on average.

5 Conclusion

In this paper, a systematic approach to minimize the number of memory accesses during neighborhood exploration has been applied to three widely employed iterative parallel thinning algorithms. The reported results clearly demonstrate that a significant improvement can be obtained on the state-of-the-art.

The availability of a public and open-source system (THeBE) allows researchers and practitioners to really test the best solutions on their specific environment, and to possibly further improve them in the future.

References

1. Agam, G., Argamon, S., Frieder, O., Grossman, D., Lewis, D.: The Complex Document Image Processing (CDIP) Test Collection Project. Illinois Institute of Technology (2006)
2. Bolelli, F.: 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)
3. Bolelli, F., Baraldi, L., Cancilla, M., Grana, C.: Connected Components Labeling on DRAGs. In: International Conference on Pattern Recognition (ICPR). pp. 121–126. IEEE (2018)
4. Bolelli, F., Borghi, G., Grana, C.: Historical Handwritten Text Images Word Spotting Through Sliding Window Hog Features. In: 19th International Conference on Image Analysis and Processing (ICIAP). pp. 729–738. Springer (2017)
5. Bolelli, F., Borghi, G., Grana, C.: XDOCS: An Application to Index Historical Documents. In: Italian Research Conference on Digital Libraries (IRCDL). pp. 151–162. Springer (2018)
6. Bolelli, F., Cancilla, M., Baraldi, L., Grana, C.: Connected Components Labeling on DRAGs: Implementation and Reproducibility Notes. In: Reproducible Research in Pattern Recognition (RRPR). pp. 89–93. Springer (2018)
7. Chaudhuri, B.B., Adak, C.: An approach for detecting and cleaning of struck-out handwritten text. Pattern Recognition **61**, 282–294 (2017)
8. Chen, Y.S., Hsu, W.H.: A modified fast parallel algorithm for thinning digital patterns. Pattern Recognition Letters **7**(2), 99–106 (1988)
9. Chen, Y.S., Hsu, W.H.: A modified fast parallel algorithm for thinning digital patterns. Pattern Recogn Lett **7**(2), 99–106 (1988). https://doi.org/DOI:10.1016/0167-8655(88)90124-9

10. Deutsch, E.S.: Thinning Algorithms on Rectangular, Hexagonal, and Triangular Arrays. Communications of the ACM **15**(9), 827–837 (1972)
11. Dinneen, G.: Programming pattern recognition. In: Proceedings of the Western Joint Computer Conference. pp. 94–100. ACM (1955)
12. Grana, C., Baraldi, L., Bolelli, F.: Optimized Connected Components Labeling with Pixel Prediction. In: Advanced Concepts for Intelligent Vision Systems (ACIVS). pp. 431–440. Springer (2016)
13. Grana, C., Borghesani, D.: Optimal decision tree synthesis for efficient neighborhood computation. In: Proceedings of the XI Conference of the Italian Association for Artificial Intelligence (AIXIA). pp. 92–101. Reggio Emilia, Italy (2009)
14. Grana, C., Borghesani, D., Cucchiara, R.: Decision Trees for Fast Thinning Algorithms. In: 20th International Conference on Pattern Recognition (ICPR). pp. 2836–2839 (2010)
15. Guo, Z., Hall, R.W.: Parallel Thinning with Two-Subiteration Algorithms. Communications of the ACM **32**(3), 359–373 (1989)
16. Hall, R.W.: Fast Parallel Thinning Algorithms: Parallel Speed and Connectivity Preservation. Communications of the ACM **32**(1), 124–131 (1989)
17. The Hamlet Dataset, http://www.gutenberg.org, accessed on 2019-05-02
18. He, L., Zhao, X., Chao, Y., Suzuki, K.: Configuration-Transition-Based Connected-Component Labeling. IEEE Transactions on Image Processing" **23**(2), 943–951 (2014)
19. He, S., Schomaker, L.: DeepOtsu: Document enhancement and binarization using iterative deep learning. Pattern Recognition **91**, 379–390 (2019)
20. Holt, C.M., Stewart, A., Clint, M., Perrott, R.H.: An Improved Parallel Thinning Algorithm. Communications of the ACM **30**(2), 156–160 (1987)
21. Huang, T., Yang, G., Tang, G.: A fast two-dimensional median filtering algorithm. IEEE Transactions on Acoustics, Speech, and Signal Processing **27**(1), 13–18 (1979)
22. Khodadoust, J., Khodadoust, A.M.: Fingerprint indexing based on minutiae pairs and convex core point. Pattern Recognition **67**, 110–126 (2017)
23. Lam, L., Lee, S.W., Suen, C.Y.: Thinning Methodologies—A Comprehensive Survey. IEEE T Pattern Anal **14**(9), 869–885 (1992). https://doi.org/http://dx.doi.org/10.1109/34.161346
24. Lewis, D., Agam, G., Argamon, S., Frieder, O., Grossman, D., Heard, J.: Building a test collection for complex document information processing. In: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval. pp. 665–666. ACM (2006)
25. Lü, H., Wang, P.S.P.: A Comment on "A Fast Parallel Algorithm for Thinning Digital Patterns". Communications of the ACM **29**(3), 239–242 (1986)
26. Maltoni, D., Maio, D., Jain, A., Prabhakar, S.: Handbook of Fingerprint Recognition. Springer Science & Business Media (2009)
27. Documentation of the thinning function in OpenCV, https://docs.opencv.org/4.0.0/df/d2d/group__ximgproc.html#ga37002c6ca80c978edb6ead5d6b39740c, accessed on 2019-05-02
28. Sauvola, J., Pietikäinen, M.: Adaptive document image binarization. Pattern recognition **33**(2), 225–236 (2000)
29. Schutte, L.J.: Survey of decision tables as a problem statement technique. CSD-TR 80, Computer Science Department, Purdue University (1973)
30. The Legacy Tobacco Document Library (LTDL). University of California, San Francisco (2007)

31. Source code of the THeBE benchmarking system, https://github.com/prittt/THeBE, accessed on 2019-05-02
32. The THeBE dataset, http://aimagelab.ing.unimore.it/files/THeBE_dataset.zip, accessed on 2019-05-02
33. Uslu, F., Bharath, A.A.: A recursive Bayesian approach to describe retinal vasculature geometry. Pattern Recognition **87**, 157–169 (2019)
34. Wang, X., Jiang, X., Ren, J.: Blood vessel segmentation from fundus image by a cascade classification framework. Pattern Recognition **88**, 331–341 (2019)
35. Zhang, T., Suen, C.Y.: A Fast Parallel Algorithm for Thinning Digital Patterns. Communications of the ACM **27**(3), 236–239 (1984)