In-place Linear-time Algorithms for Euclidean Distance Transform

Tetsuo Asano and Hiroshi Tanaka

School of Information Science, JAIST,
1-1 Asahidai, Nomi, 923-1292, Japan.

Abstract. Given a binary image, Euclidean distance transform is to compute for each pixel the Euclidean distance to the closest black pixel. This paper presents a linear-time algorithm for Euclidean distance transform without using any extra array. This is an improvement over a known algorithm which uses additional arrays as work storage. An idea to reduce the work space is to utilize the output array as work storage. Implementation results and comparisons with existing algorithms are also included.

1 Introduction

There are increasing demands for highly functional peripherals such as printers, scanners, and digital cameras. To achieve intelligence they need sophisticated built-in or embedded softwares. One big difference from ordinary software in computers is little allowance of working space which can be used by the software. In the sense space-efficient algorithms are requested. A number of in-place algorithms have been studied [1–4, 6]. In this paper, we propose another in-place algorithm used for image processing. Especially, we present an algorithm for Euclidean distance transform, which is one of the central problems in image processing. Given a binary image, a distance transform algorithm computes for each pixel how close it is to the closest black pixel.

This paper presents a linear-time in-place algorithm for Euclidean distance transform without using any extra array in addition to a given image matrix. The Euclidean distance transform is to compute a distance map in which each element is the Euclidean distance from the element (pixel) to the closest black element. A number of different algorithms have been proposed and implemented for the Euclidean distance transform [5, 7, 9, 8, 10, 11]. It has been widely applied in many different problems, especially in medical image analysis [12–14].

2 Distance Transform

Distance Transform is one of the most important operations in image processing. Given a binary image G, our target is to compute a matrix D of the same size such that each element $D(x, y)$ is the distance from each corresponding pixel (x, y) to the closest black pixel (including itself). A simple example of

a binary image is given in Figure 1(a) in which black pixels and white pixels are expressed by black disks and white circles, respectively. The closest black pixel from each white pixel is indicated by an arrow. Figure 1(b) is a matrix of distance transform in which each element is the distance transformation value, the Euclidean distance from the corresponding pixel to the closest black pixel. Note that the distance is 0 for every black pixel.

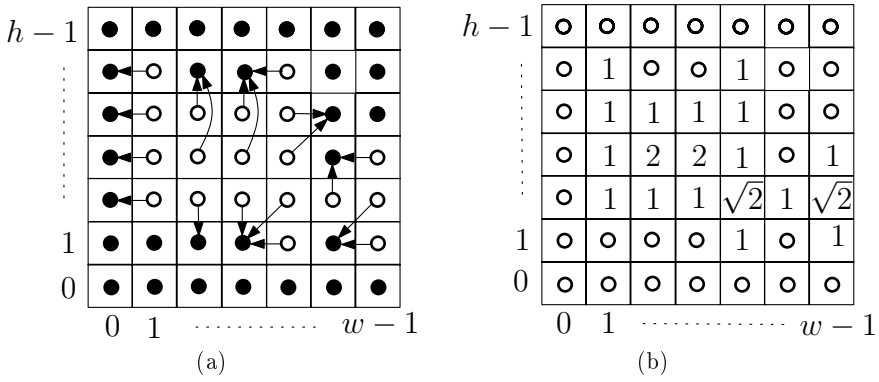

Fig. 1. Euclidean distance transform. (a) Closest black pixel (black disk) from each white pixel (white circle) in an binary image G. (b) The corresponding distance matrix D.

A brute-force algorithm for distance transform is a breadth-first search simulating a wave propagation. Starting from a white pixel, we propagate a wave in the increasing way of the distances from the starting pixel until it encounters some black pixel. It is easy to implement using a queue, but unfortunately, it takes quadratic time in the worst case.

3 Known Algorithms

It was open for many years whether the Euclidean distance transform [8] can be computed in linear time or not, but it was finally solved in an affirmative way in 1995 and 1996 by two different groups [10, 11]. They are based on different ideas, but both run in linear time in the number of pixels. In this paper we implement the algorithm [11] by Hirata et al. using constant amount of working space.

Their algorithm is described in a pseudo-code as follows.

Input: $h \times w$ bit array G for an input binary image with $n = hw$ pixels.
array: $h \times w$ array D of squared distances.
Phase 1: Vertical Scan
  for each $x_c = 0$ **to** $w - 1$
    $d = 0$.

```
    for each y_c = 1 to h − 1
        if G(x_c, y_c) > 0 then d = d + 1 else d = 0.
        D(x_c, y_c) = d.
  for each x_c = 0 to w − 1
    d = 0.
    for each y_c = h − 1 to 0
        if G(x_c, y_c) > 0 then d = d + 1 else d = 0.
        D(x_c, y_c) = min{D(x_c, y_c), d}.
Phase 2: Horizontal Scan
  for each y_c = 0 to h − 1
    Initialize a stack.
    Push (0, D(0, y_c)) and (1, D(1, y_c)) into the stack.
    for x_c = 2 to w − 1
      repeat_forever{
        (X_s, Y_s) = the rightmost intersection among parabolas in the stack.
        if Y_s ≤ (X_s − x_c)^2 + D(x_c, y_c)^2 then exit the loop.
        else remove the top element out of the stack.
      }
      Push (x_c, D(x_c, y_c)) into the stack.
    for x_c = w − 1 down to 0
      repeat_forever{
        (X_s, Y_s) = the rightmost intersection among parabolas in the stack.
        if X_s ≤ x_c then exit the loop.
        else remove the top element out of the stack.
      }
      (x_t, g_t) = top_element(stack).
      D(x_c, y_c) = (x_c − x_t)^2 + g_t^2.
        // squared distance to the closest black pixel
```

The algorithm is divided into two phases. The first phase is to compute the vertical distance to the closest black pixel by two vertical scans, one from bottom to top and the other from top to bottom, while keeping the one-directional distances from black pixel. If we take the smaller distance at each pixel, it is the vertical distance required.

At the beginning of the second phase the vertical distances are calculated in the matrix D. This phase consists of two horizontal scans, one from left to right and the other from right to left. The idea to compute the Euclidean distance is the following. A value $D(x_c, y_c)$ means that the vertically closest black black pixel from the pixel at (x_c, y_c) is either at $(x_c, y_c - D(x_c, y_c))$ or $(x_c, y_c + D(x_c, y_c))$, both in the distance $D(x_c, y_c)$ from (x_c, y_c). It is not important whether the black pixel is above or below it, but the distance $D(x_c, y_c)$ is meaningful. The black pixel at $(x_c, y_c \pm D(x_c, y_c))$ may be the closest black pixel for a different white pixel at (x_s, y_c) in the same row. Its squared distance is given by $(x_c - x_s)^2 + D(x_c, y_c)^2$. So, if we define a parabola $y = (x - x_c)^2 + D(x_c, y_c)^2$ for each pixel (x_c, y_c) in the row $y = y_c$ then the squared distance from a pixel (x_c, y_c) to the closest black pixel is equal to the lower envelope of those parabolas.

It is rather easy to compute the lower envelope of those parabolas since every parabola has the same shape and thus any two intersect at a single point. We construct the lower envelope for the row $y = y_c$ using a stack as follows. The stack initially consists of two elements $(0, D(0, y_c))$ and $(1, D(1, y_c))$, which define parabolas $y = (x - x_c)^2 + D(x, y_c)^2$, $x = 0, 1$. Then, for each x_c from 2 to $n - 1$ we take the top two elements (parabolas) from the stack and compute their intersection (X_s, Y_s) that is the rightmost intersection among parabolas in the stack in the algorithm above. It is indeed easy to compute the intersection of two parabolas, $P_a : y = (x - x_a)^2 + D(x_a, y_c)^2$ and $P_b : y = (x - x_b)^2 + D(x_b, y_c)^2$. If the intersection (X_s, Y_s) lies above the current parabola $P_c : y = (x - x_c)^2 + D(x_c, y_c)^2$, that is, if $Y_s > (X_s - x_c)^2 + D(x_c, y_c)^2$, then the top element (parabola) in the stack never contributes to the lower envelope since the right part of the parabola P_b lies above the current parabola P_c and its left part is above P_a. That is why the top element of the stack must be removed. In this way we remove top elements of the stack until the rightmost intersection associated with the stack lies below the current parabola. Then, we push the current parabola into the stack.

Once we have constructed the lower envelope using the stack, we scan the same row of the matrix D from right to left, that is, from $x_c = w - 1$ down to $x_c = 0$. At each pixel (x_c, y_c) in the row, we want to compute the vertical distance to the lower envelope. Since the envelopes in the stack are arranged in the increasing order of the x coordinates of their peaks, we can take out those parabolas in the decreasing order of their x-coordinates by popping up the stack. If the parabola just above a pixel (x_c, y_c) is $y = (x - x_t)^2 + g_t^2$ then the vertical distance to the lower envelope is given by $(x_c - x_t)^2 + g_t^2$ (see Figure 2).

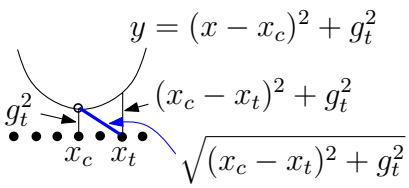

Fig. 2. Influence of the white element closest from a black element over other black elements in the same row.

Theorem 1 (Hirata et al. [11]). *There is a linear-time algorithm for computing the distance transform for a binary image.*

4 Constant-Working-Space Algorithm

The algorithm given above is not space-efficient. It uses two working spaces, one for distance matrix D of size $n = h \times w$ and the other for a stack of size $\max\{h, w\}$. It is rather straightforward to reduce the space for the matrix D.

It suffices to use the input binary matrix as D. Just rename D as G, the input image matrix in the algorithm. It causes no problem at all since all the black pixels remain 0 and all the white pixels have positive distances.

It is not so straightforward to reduce the space for the stack. An idea in this paper is to use the distance matrix itself as the stack. When we are computing the distance transform values in the row $y = y_c$, the matrix elements $D(0, y_c), \ldots, D(n-1, y_c)$ keep the vertical distances to the closest black pixels. First of all we push two elements $(0, D(0, y_c))$ and $(1, D(1, y_c))$ into the stack. Instead of using the stack, we put them at $D(0, y_c)$ and $(0, D(0, y_c))$, that is, we set

$$\begin{aligned} D(0, y_c) &= 0 + w \times D(0, y_c), \text{ and} \\ D(1, y_c) &= 1 + w \times D(1, y_c). \end{aligned}$$

Of course, the original values $D(0, y_c)$ and $D(1, y_c)$ are lost, but they can be restored by dividing the new value $D(0, y_c)$ and $D(1, y_c)$, and then taking their integral parts. We maintain the number of stack elements by a variable, say t. Whenever we want to push an element $(x_c, D(x_c, y_c))$ into the stack, we simulate it by setting

$$\begin{aligned} D(t, y_c) &= x_c + w \times D(x_c, y_c), \\ t &= t + 1 \quad \text{(increment the value of } t\text{)}. \end{aligned}$$

Removing the top element out of the stack is easy. Just decrement the value of t. Whenever we need an element of a stack, we take the top element $D(t-1, y_c)$. Then, the corresponding x and $D(x, y_c)$ values can be calculated by

$$\begin{aligned} x &= D(t-1, y_c) \mod w, \text{ and} \\ D(x, y_c) &= \lfloor D(t-1, y_c)/w \rfloor. \end{aligned}$$

An important observation is that once we have constructed a stack, the original values of the distance matrix D are not needed anymore. One thing we have to worry about is that whenever we compute the vertical distance at $x = x_c$ we can safely store the value at the corresponding matrix element, $D(x_c, y_c)$. In other words, whenever we store the vertical distance at $D(x_c, y_c)$, the element $D(x_c, y_c)$ must not be a part of the stack. If it is a necessary element of the stack, then it destroys the stack.

It is not so simple to guarantee the safeness. To easily convince ourselves of the safeness we apply an operation called **redundancy removal** to the resulting stack. The operation of redundancy removal is to remove redundant elements from stack. Let $(x_0, g_0), (x_1, g_1), \ldots, (x_{t-1}, g_{t-1})$ be a content of a stack. Each element (x_i, g_i) corresponds to a parabola

$$P_i : y = (x - x_i)^2 + g_i^2, \ i = 0, 1, \ldots, t-1. \tag{1}$$

Any two consecutive parabolas P_i and P_{i+1} intersect at a single point, which is denoted by (X_i, Y_i). Now, we have some basic observations.

Observation 1: $(x_0, x_1, \ldots, x_{t-1})$ is an increasing sequence, that is, $x_0 < x_1 < \cdots < x_{t-1}$.

Proof. Immediate from the construction of the stack.

Observation 2: A sequence of intersections $(X_0, X_1, \ldots, X_{t-2})$ is also an increasing sequence, that is, $X_0 < X_1 < \cdots < X_{t-2}$.

Proof. For contradiction suppose $X_{i-1} > X_i$. By the definition, (X_{i-1}, Y_{i-1}) is the intersection of two parabolas $P_{i-1} : y = (x - x_{i-1})^2 + g_{i-1}^2$ and $P_i : y = (x - x_i)^2 + g_i^2$. If the third parabola $P_{i+1} : y = (x - x_{i+1})^2 + g_{i+1}^2$ passes through the intersection (X_{i-1}, Y_{i-1}) then we have $X_i = X_{i-1}$, which never happens since it means P_i coincides with P_{i+1}. Thus, the inequality $X_{i-1} > X_i$ implies that the point (X_{i-1}, Y_{i-1}) lies above the parabola P_{i+1}. If it is true, then the parabola P_i must have been removed from the stack when P_i is pushed into the stack. This is a contradiction requested.

Now we define redundant stack elements. As is stated before, the stack gives a sequence of parabolas appearing in the lower envelope. An important notice here is that our goal is not to compute the lower envelope but the vertical distance from each pixel in the current row to the lower envelope. Since each pixel had an integral x-coordinate, a stack element is meaningful only if its corresponding parabola appears in the lower envelope at some integral x-coordinate. Otherwise, that is, if the interval in which a parabola P_i appears in the lower envelope contains no integral x-coordinate, the parabola P_i is useless or **redundant**, which can be safely deleted from the lower envelope (or from the stack) without affecting the calculation of vertical distances.

Observation 3: If $X_0 < 0$ then the bottom element (x_0, g_0) in the stack is redundant.

Proof. If $X_0 < 0$ holds, then the two parabolas P_0 and P_1 in the stack intersect at some point to the left of $x = 0$. This means that the parabola P_0 lies above P_1 in the x-interval $[0, w - 1]$. This means that the parabola P_0 never appears in the lower envelope in the x-interval.

Observation 4: If $\lfloor X_{i-1} \rfloor + 1 = \lceil X_i \rceil$ then the element (x_i, g_i) in the stack is redundant.

Proof. Recall that (X_{i-1}, Y_{i-1}) (resp. (X_i, Y_i)) is intersection of two consecutive parabolas P_{i-1} and P_i (resp., P_i and P_{i+1}). The equation $\lfloor X_{i-1} \rfloor + 1 = \lceil X_i \rceil$ means that the x-coordinates of the two intersections are between two consecutive integers as shown in Figure 3. This means that the middle parabola does not contribute to the lower envelope at any integral x-coordinate.

Based on the observations above, we remove all redundant elements from the stack. The first kind of redundant stack elements can be removed by repeatedly applying Observation 3 so that the bottom element gives the lower envelope at $x = 0$. The second kind of redundant stack elements are those which have no

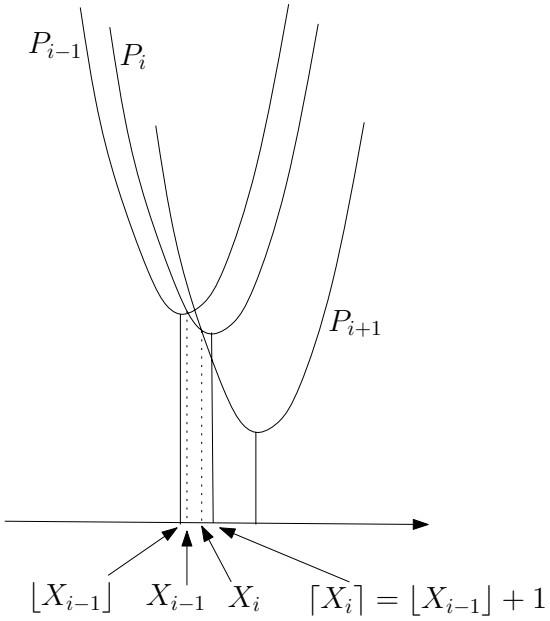

Fig. 3. Three parabolas and their intersections.

contribution at integral x-coordinate. We check two consecutive stack elements and if $\lfloor X_{i-1} \rfloor + 1 = \lceil X_i \rceil$ holds then we remove one stack element as described in Observation 4.

Now the algorithm using only constant amount of working space is given as follows.

Constant-Working-Space Algorithm for Euclidean Distance Transform
Input: $h \times w$ binary image G with $n = hw$ pixels.
Phase 1: Vertical Scan
  for each $x_c = 0$ **to** $w - 1$
    $d = 0$.
    for each $y_c = 1$ **to** $h - 1$
      if $G(x_c, y_c) > 0$ then $d = d + 1$ else $d = 0$.
      $G(x_c, y_c) = d$.
  for each $x_c = 0$ **to** $w - 1$
    $d = 0$.
    for each $y_c = h - 1$ **to** 0
      if $G(x_c, y_c) > 0$ **then** $d = d + 1$ **else** $d = 0$.
      $G(x_c, y_c) = \min\{G(x_c, y_c), d\}$.
Phase 2: Horizontal Scan
  for each $y_c = 0$ **to** $h - 1$
    $G(0, G(0, y_c)) = 0 + w \times G(0, y_c)$.
    $G(1, G(1, y_c)) = 1 + w \times G(1, y_c)$.

$t = 2$ // Initialize a stack with two elements.
for $x_c = 2$ **to** $w - 1$
repeat_forever{
(X_{t-2}, Y_{t-2}) = the rightmost intersection among parabolas in the stack.
if $Y_{t-2} \leq (X_{t-2} - x_c)^2 + G(x_c, y_c)^2$ **then** exit the loop.
else $t = t - 1$ (remove the top element out of the stack).
}
$G(t, y_c) = x_c + w \times G(x_c, y_c)$, $t = t + 1$.
// Push $(x_c, D(x_c, y_c))$ into the stack.

Phase 3: Redundancy Removal.
Remove the top element of the stack while it does not contribute to the lower envelope at $x = 0$.
Remove any stack element whose corresponding parabola has no contribution to the lower envelope at any integral x-coordinate.

Phase 4: Final Distance Calculation.
for $x_c = w - 1$ **down to** 0
repeat_forever{
(X_{t-2}, Y_{t-2}) = the rightmost intersection among parabolas in the stack.
if $X_{t-2} \leq x_c$ **then** exit the loop.
else $t = t - 1$ (remove the top element out of the stack).
}
$x_a = G(t-1, y_c) \mod n$, $g_a = \lfloor G(t-1, y_c)/w \rfloor$
// top element of the stack.
$G(x_c, y_c) = (x_c - x_a)^2 + g_a^2$. // the squared Euclidean distance

Theorem 2. *Given a binary image G with n pixels, the algorithm above computes the Euclidean Distance Transform in $O(n)$ time without using any extra array, that is, it outputs a distance matrix of the same size as that of the input image such that each element is the distance from the corresponding pixel to its closest black pixel. Moreover, the largest possible integer stored in the distance matrix during the implementation of the algorithm is bounded by $(n-1)^2$.*

Proof. It is clear that the algorithm does not use any extra array in addition to an input image array and one for distance map. It is also clear that the largest possible integer stored in the matrix is at most $(n-1)^2$, which is greater than the squared distance between two corner pixels of an input image with n pixels. The correctness of the algorithm follows from the fact that the squared Euclidean distance from each pixel p to its closest black pixel q is given by the squared sum of their horizontal distance (defined by their y-coordinates) and their horizontal distance (defined by their x-coordinates), which is equal to the vertical distance between the pixel p and the parabola defined by the pixel q. The algorithm consists of double loops, but in each iteration of the inner loop the top element of the stack is removed. Since any pixel is popped more than twice, the total number of iterations in the double loops is bounded by the total number of pixels, that is, $O(n)$.

In the algorithm above we implement a stack using the input matrix itself. The output is a matrix with each element being the (squared) Euclidean distance to its closest black pixel. The largest possible squared Euclidean distance is $(n-1)^2$, which is achieved by a $1 \times n$ array $100\cdots00$. Thus, the matrix element must have at least $2\log_2(n-1)$ bits.

To implement the stack, we put x-coordinate x_c and a vertical distance d in a packed manner, that is, $x + w \times d \leq n + w \times h \leq 2n$, which is less than the largest possible value $(n-1)^2$ in the output matrix.

5 Bidirectional Distance Transform

Now, consider an extension to a bidirectional Euclidean Distance Transform in which we are requested to compute for each pixel p in a given binary image the distance to the closest pixel with a different value from p, that is, distances in both directions from 0 to 1 and from 1 to 0 are required. The bidirectional Euclidean distance transform is rather straightforward if $O(n)$ working space is available. Our goal here is to compute the distance maps using only constant amount of working space. To distinguish distances from 0 to 1 and from 1 to 0, we use signs. For each white pixel the distance is positive, but it is negative for each black pixel.

Idea is the following. In the first two scans we compute vertical distances as before. It is easy to adapt the vertical scan described before to compute bidirectional vertical distances. The resulting distances are stored over the input binary image as before but with signs to distinguish between black and white pixels. The resulting value $d(x, y)$ means existence of opposite-valued pixel in the vertical distance $|d(x, y)|$. If (x, y) is originally white pixel, then $d(x, y)$ is positive. Otherwise, $d(x, y)$ has the minus sign.

Our algorithm is quite similar to the previous one. Suppose we are processing a row $y = y_c$. If the row contains no black pixel, that is, every vertical distance in the row is positive, then we can use the same algorithm as before. So, suppose we have a number of 0-runs and 1-runs in the row. Let $[s, t]$ be an interval of a maximal 1-run in the row. Without loss of generality we assume that

$p(s-1, y_c) = 0, p(x, y_c) = 1,$ for $x = s, \ldots, t,$ and $p(t+1, y_c) = 0.$

Here, $p(x, y)$ denotes the original pixel value (0 or 1) at (x, y) in the original binary image. Then, the distance value at any pixel $p(x, y_c)$ in this interval is not affected by vertical distances at pixels in the row in the exterior of the interval. This means that we can compute distance values run by run. An important observation is that a stack fits into the interval of the run (possibly using two more elements corresponding to the black pixels in both ends).

Theorem 3. *Given a binary image G with n pixels, we can compute without using any extra array a distance map d in linear time over G such that*
(1) $d(x, y) > 0$ means that the pixel (x, y) is white pixel and the distance to the closest black pixel is $d(x, y)$, and
(2) $d(x, y) < 0$ means that the pixel (x, y) is black pixel and the distance to the closest white pixel is $-d(x, y)$.

6 Concluding Remarks and Future Works

In this paper we have presented a linear-time algorithm for Euclidean Distance Transform which uses only constant amount of working space. We have implemented our algorithm and verified that the running time is almost the same as that of the existing algorithm using stack. An advantage of the proposed algorithm is space-efficiency. It is important for applications in embedded software for visual equipments such as digital cameras.

One of the future works is to extend the idea in this paper to a multiple intensity image. Given an image G with intensities $0, 1, \ldots, g$ for some constant $g \geq 1$, for each pixel (x, y) with intensity $G(x, y) \geq 1$ we want to compute the distance to a closest pixel with intensity less than $G(x, y)$, and also for each pixel (x, y) with intensity $G(x, y) < g$ we want to compute the distance to a closest pixel with intensity greater than $G(x, y)$.

Acknowledgments

This work of T.Asano was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research on Priority Areas and Scientific Research (B). The author would like to express his sincere thanks to Erik Demaine, Stefan Langerman, Ryuhei Uehara, Mitsuo Motoki, and Masashi Kiyomi for their stimulating discussions.

References

1. T. Asano, S. Bitou, M. Motoki and N. Usui,"Space-Efficient Algorithm for Image Rotation," IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol.91-A(9), pp.2341-2348, 2008.
2. T. Asano, "Constant-Working-Space Image Scan with a Given Angle," Proc. 24th European Workshop on Computational Geometry, Nancy, France, pp.165-168, 2008.
3. T. Asano, "Constant-work-Space Algorithms: How Fast Can We Solve Problems without Using Any Extra Array?," Invited talk at ISAAC 2008, p.1, Dec. 2008.
4. T. Asano, "Constant-work-Space Algorithms for Image Processing," Monograph: "ETVC08: Emerging Trends and Challenges in Visual Computing," ETVC 2008: pp.268-283, edited by Frank Nielsen, 2009.
5. G. Borgefors, "Distance transformations in digital images," Computer Vision, Graphics and Image Processing, 34, pp.344-371, 1994.
6. T. M. Chan, E. Y. Chen, "Multi-Pass Geometric Algorithms," Discrete & Computational Geometry 37(1), pp.79-102, 2007.
7. L. Chen and H.Y.H. Chuang, "A fast algorithm for Euclidean distance maps of a 2-D binary image," Information Processing Letters, 5, 1, pp.25-29, 1994.
8. R. Fabbri, L. da F. Costa, J. C. Torelli, and O. M. Bruno, "2D Euclidean Distance Transform Algorithms: A Comparative Survey," ACM Computing Surveys, 40, 1, pp:2:1–2:44, 2008.
9. M.L. Gabrilova and M. Alsuwaiyel, "Computing the Euclidean distance transform," Journal of Supercomputing, 25, pp.177-185, 2003.

10. H. Breu, J. Gil, D. Kirkpatrick, and M. Werman, "Linear Time Euclidean Distance Algorithms," IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol.17, No.5, pp.529-533, 1995.
11. T. Hirata, "A unified linear-time algorithm for computing distance maps," Information Processing Letters, Vol.58, No.3, pp.129-133, 1996.
12. F. Klein and O. Kübler, "Euclidean distance transformations and model guided image interpretation," Pattern Recognition Letters, 5, pp.19-20, 1987.
13. D.W. Paglieroni, "Distance Transforms," Computer Vision, Graphics and Image Processing: Graphical Models and Image Processing, 54, pp.56-74, 1992.
14. A. Rosenfeld and A.C. Kak, "Digital Picture Processing," Academic Press, New York, second edition, 1978.