The Distance Transform and its Computation — An Introduction —

©Tilo Strutz
Technical paper, June, 2021, TECHP/2021/06
Leipzig University of Telecommunication

Abstract

Distance transformation is an image processing technique used for many different applications. Related to a binary image, the general idea is to determine the distance of all background points to the nearest object point (or vice versa). In this tutorial, different approaches are explained in detail and compared using examples. Corresponding source code is provided to facilitate own investigations. A particular objective of this tutorial is to clarify the difference between arbitrary distance transforms and **exact Euclidean** distance transformations.

1 General description

1.1 Examples and possible applications

Figure 1 shows what can be achieved in general by a distance transform. The input is typically a binary image (Figure 1a), i. e. the pixels can have only one out of two different values. One value is associated to the background and the other value defines the object pixels. The object pixels can belong to one or more (disconnected) objects. Depending on the application, either the distances inside the objects are of interest (Figure 1b) or the distances outside the objects (Figure 1c).

The segmentation of images in regions is one important application of the distance transform. Afterwards, each region represents one object point or a cluster of connected points. This is highly related to so-called Voronoi diagram. **Figure 2** and **Figure 3** shows two examples. From these Voronoi diagrams it can be deduced which points are adjacent. This information could be helpful in solving graph-based problems.

The distance within objects also could be of interest for finding the centre of objects like hands or for the skeletonization of objects, see **Figure 4**. Intermediate processing results may vary when different distance metrics (which are discussed in Subsection 1.3) are used, see **Figure 5**. And last but not least, distance transforms can support the segmentation of overlapping objects like cells, **Figure 6**.

This paper wont discuss, how the distance transform has to be used in achieving the goals of these applications; instead, it concentrates on the basics of distances and on algorithms for computing a distance transformation in an appropriate manner. First ideas about distances in digital images have been discussed in a pioneering work by Rosenfeld and Pfaltz in 1968 [Ros68].

The following two Subsections discuss neighbourhood relations and different metrics for distances. If you are already familiar with it, just skip to Section 2.

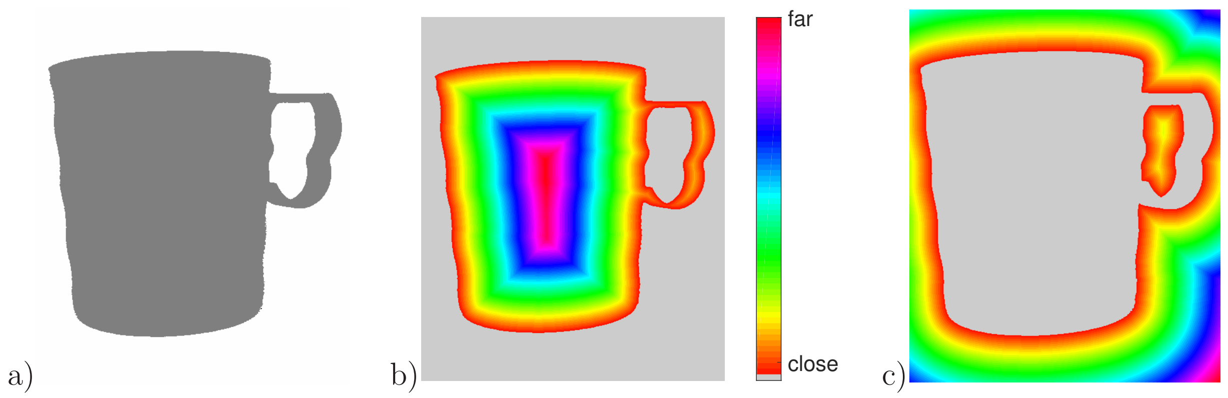


Figure 1: Example of a distance transform: a) binary image containing a dark object on white background; b)+c) results of distance transform showing the distance of object pixels to closest background pixel (b) and the distance of background pixels to closest object pixel (c)

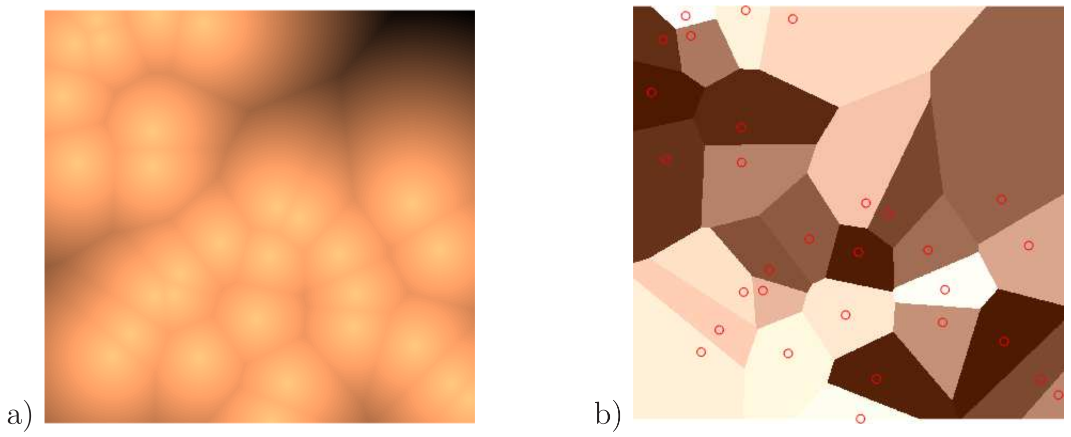


Figure 2: Example image with 30 separate object points: a) distance matrix; b) Voronoi diagram with marked positions of object points.

1.2 Neighbourhood relations

In an orthogonal grid of pixels, we can differentiate between two neighbourhood relations. The first is called 4-neighbourhood since only the horizontal and vertical neighbouring pixels are considered. When also the diagonal pixels are taken into account it is called 8-neighbourhood, **Figure 7** a).

This differentiation has implications in several directions. If, for example, binary image objects have to be counted, the result can be different. **Figure 7** b) shows dark object pixels, which belong to a single object. In 8-neighbourhood, all pixels belong to a single object, because the marked pixels are considered as neighbours. However, in 4-neighbourhood the marked pixels are not connected and there are two binary objects.

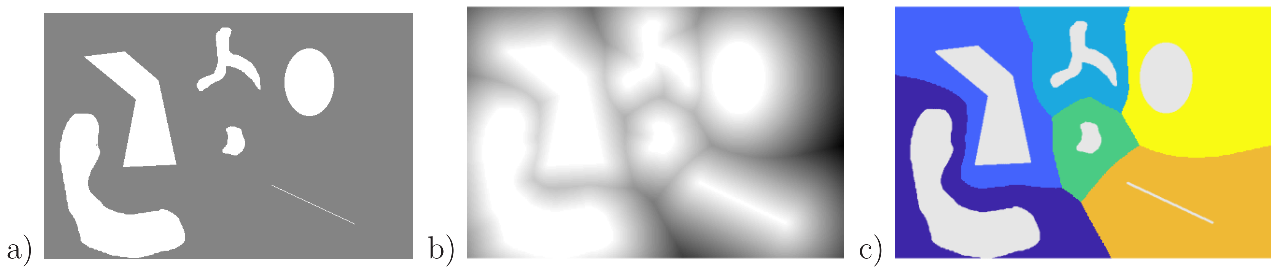


Figure 3: Example image containing six binary objects: a) original binary image; b) distance matrix; c) Voronoi diagram.

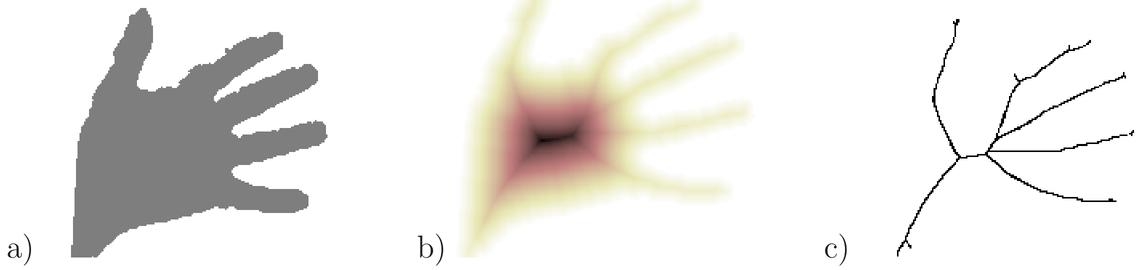


Figure 4: Steps in hand-gesture recognition: a) original binary image; b) distance matrix for finding the centre of the palm; c) derived skeleton.

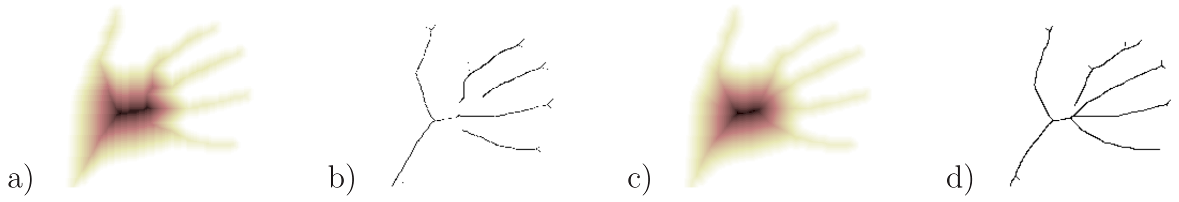


Figure 5: Results for non-Euclidean distances: a) distance matrix based on city-block distance; b) derived skeleton; c) distance matrix based on chamfer-43 distance; d) derived skeleton.

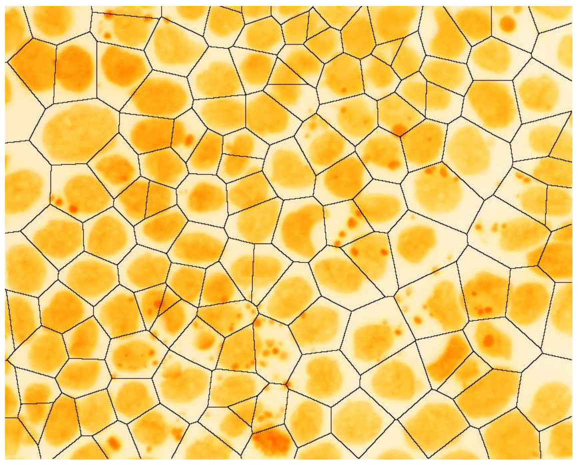


Figure 6: Separation of objects utilizing distance transforms.

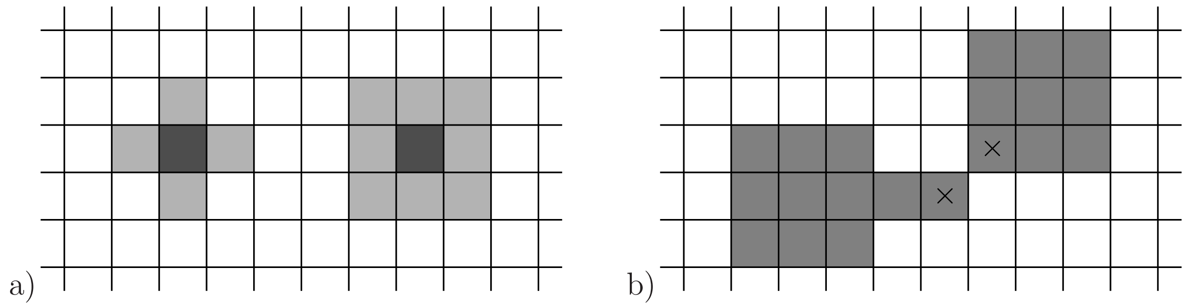
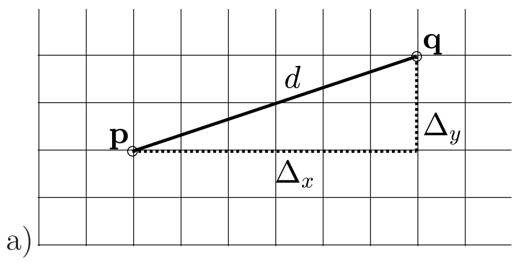


Figure 7: Neighbourhood configurations: a) 4-neighbourhood (left) and 8-neighbourhood (right); b) dark object pixels are considered as a single object in 8-neighbourhood but as two (not connected) objects in 4-neighbourhood.



b)

32	25	20	17	16	17	20	25	32	41
25	18	13	10	9	10	13	18	25	34
20	13	8	5	4	5	8	13	20	29
17	10	5	2	1	2	5	10	17	26
16	9	4	1	**0**	1	4	9	16	25
17	10	5	2	1	2	5	10	17	26
20	13	8	5	4	5	8	13	20	29
25	18	13	10	9	10	13	18	25	34
32	25	20	17	16	17	20	25	32	41

Figure 8: Euclidean distance in two dimensions: a) $\Delta_x = q_x - p_x$, $\Delta_y = q_y - p_y$, the distance is $d = \sqrt{\Delta_x^2 + \Delta_y^2}$; b) calculated squared distances d^2 from a centre pixel.

1.3 Distance Metrics

Minkowski metric

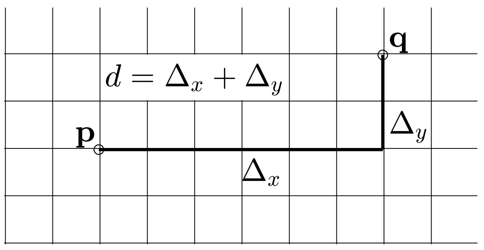
The distances shown in Figure 1 are based on the Euclidean metric. This is what we also know as beeline distance. If we hammer two nails into a board and connect them with a rubber band, then the Euclidean distance is equal to the one-way length of the rubber band. In two dimensions this distance is defined by the Pythagorean theorem: $c^2 = a^2 + b^2$. If there are two points $\mathbf{p}$ and $\mathbf{q}$ with two-dimensional coordinates (p_x, p_y) and (q_x, q_y), then the Euclidean distance d is computed with $d = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$, see **Figure 8**.

Depending on the application, different metrics could be useful. A more general definition is given by the Minkowski distance:

$$d(\mathbf{p}, \mathbf{q}) = \left[\sum_{i=1}^{D} |p_i - q_i|^e \right]^{1/e} \tag{1}$$

with D being the number of dimensions. The Euclidean metric is a special case with $e = 2$. Alternatively, $e = 1$ corresponds to the city-block distance, see **Figure 9**. This distance is sometimes called the Manhattan distance because most streets in Manhattan are either parallel or orthogonal to each other. If somebody wants to walk from point $\mathbf{p}$ to point $\mathbf{q}$, then she has to go along the streets and there is no possibility to cross the buildings.

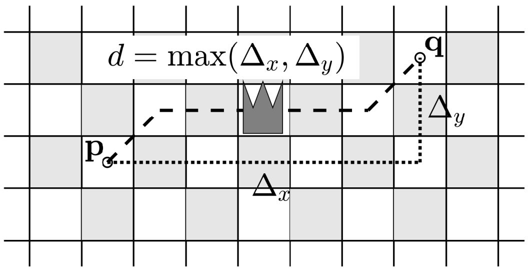
Setting variable e to infinity defines the chessboard distance. This is the number of moves that the king has to take on the chessboard from one square to another, **Figure 10**.

a)


b)

8	7	6	5	4	5	6	7	8	9
7	6	5	4	3	4	5	6	7	8
6	5	4	3	2	3	4	5	6	7
5	4	3	2	1	2	3	4	5	6
4	3	2	1	**0**	1	2	3	4	5
5	4	3	2	1	2	3	4	5	6
6	5	4	3	2	3	4	5	6	7
7	6	5	4	3	4	5	6	7	8
8	7	6	5	4	5	6	7	8	9

Figure 9: City-block distance in two dimensions: a) $\Delta_x = q_x - p_x$, $\Delta_y = q_y - p_y$, distance with $d = \Delta_x + \Delta_y$; b) calculated distances from a centre pixel.

a)


b)

4	4	4	4	4	4	4	4	4	5
4	3	3	3	3	3	3	3	4	5
4	3	2	2	2	2	2	3	4	5
4	3	2	1	1	1	2	3	4	5
4	3	2	1	**0**	1	2	3	4	5
4	3	2	1	1	1	2	3	4	5
4	3	2	2	2	2	2	3	4	5
4	3	3	3	3	3	3	3	4	5
4	4	4	4	4	4	4	4	4	5

Figure 10: Chessboard distance in two dimensions $\Delta_x = q_x - p_x$, $\Delta_y = q_y - p_y$: a) distance with $d = \max(\Delta_x, \Delta_y)$; b) calculated distances from a centre pixel.

Chamfer metric

Apart form metrics that can be derived from the Minkowski distance in (1), there are other possibilities to define distances. One is called 'chamfer' distance and it is constructed by using a different step size in diagonal directions compared to horizontal or vertical direction. Both, the city-block and the chessboard distances apply steps of '1' in horizontal and vertical direction. However, the diagonal step is equal to '2' in the former and equal to '1' in the latter case as can be seen in Figures 9 b) and 10 b).

As the Euclidean distance would fit many applications best, there have been made lots of attempts to approximate this metric by simpler constructions without the need of multiplications and drawing the square root. So, one of the first ideas was to use a diagonal step of $\sqrt{2}$. The resulting effect can be seen in **Figure 11** a). The distance values directly close to the centre pixel, in horizontal, vertical, and diagonal direction are now exactly the same as for the Euclidean distance (Figure 8b). At other positions, the distances differ. For example, the distance measure at the pixel in the bottom right corner deviates from the Euclidean distance by $\Delta d = \sqrt{44.3} - \sqrt{41}$. Borgefors has derived in [Bor84] that the average and the maximum approximation error can be minimized by using a diagonal step of 1.351. That means, the optimal relation between diagonal and other steps is

$$\frac{\text{step}_d}{\text{step}_{h,v}} = \frac{1.351}{1} \approx \frac{1.\overline{3}}{1} = \frac{4}{3} \qquad (2)$$

Using the approximation 4/3, the distance computations can now be performed in integer arithmetic if all values are scaled by factor 3. **Figure 11** b) shows the corresponding squared

a)

32	27.5	23.3	19.5	16	19.5	23.3	27.5	32	44.3
27.5	18	14.7	11.7	9	11.7	14.7	18	27.5	39.0
23.3	14.7	8	5.8	4	5.8	8	14.7	23.3	34.0
19.5	11.7	5.8	2	1	2	5.8	11.7	19.5	29.3
16	9	4	1	**0**	1	4	9	16	25
19.5	11.7	5.8	2	1	2	5.8	11.7	19.5	29.3
23.3	14.7	8	5.8	4	5.8	8	14.7	23.3	34.0
27.5	18	14.7	11.7	9	11.7	14.7	18	27.5	39.0
32	27.5	23.3	19.5	16	19.5	23.3	27.5	32	44.3

b)

28.4	25	21.8	18.8	16	18.8	21.8	25	28.4	40.1
25	16	13.4	11.1	9	11.1	13.4	16	25	36
21.8	13.4	7.1	5.4	4	5.4	7.1	13.4	21.8	32.1
18.8	11.1	5.4	1.8	1	1.8	5.4	11.1	18.8	28.4
16	9	4	1	**0**	1	4	9	16	25
18.8	11.1	5.4	1.8	1	1.8	5.4	11.1	18.8	28.4
21.8	13.4	7.1	5.4	4	5.4	7.1	13.4	21.8	32.1
25	16	13.4	11.1	9	11.1	13.4	16	25	36
28.4	25	21.8	18.8	16	18.8	21.8	25	28.4	40.1

Figure 11: Calculated (and rounded) squared chamfer distances d^2 from a centre pixels. Exact values are shown as integers: a) with diagonal steps of $\sqrt{2}$; b) with diagonal steps of 4/3.

distances which are now closer to the squared Euclidean distances on average. The already mentioned pixel in the bottom-right corner has now a distance that is closer to the Euclidean one ($|\Delta d_{43}| = \left|\sqrt{40.1} - \sqrt{41}\right| < \sqrt{44.3} - \sqrt{41}$). However, the reduced diagonal step ($1.\overline{3}$ instead of $\sqrt{2} \approx 1.41$) leads to higher absolute differences at all positions that are directly diagonal to the centre pixel.

The values have been calculated using the code listed in **Listing 1** (see appendix).

2 Non-Euclidean Distance Transforms

This section explains several versions of distance transformations that can be implemented by fast algorithms but are not able to produce exact Euclidean distances. If you are only interested in exact Euclidean distance transformations, then you can simply skip to the section 3.

In the following, the case shown in Figure 1 c) is considered. That is, the shortest distances between a background pixel and any object pixel are searched for.

2.1 City-block distance transform

One of the earliest ideas to perform a distance transformation was based on distance propagation. Starting from the positions of the object pixels, whose distances are zero by definition, the distances of the background pixels are determined step by step while incrementing the distance value. This can be achieved by sequentially processing all image rows first from top-left to bottom-right and then in backward direction.

At first, the distance image (or matrix) has to be initialized with values of zero at the positions of object pixels and with sufficient large value (larger than the maximum possible distance) at all other positions. An example is given in **Figure 12** a). Then all rows are scanned from left to right. If the distance $d(i,j)$ value at the current position is larger than the predecessor plus one $d(i,j-1)+1$, it is set to a new value $d(i,j) := d(i,j-1)+1$. Afterwards,

a)

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	0	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	0	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	0	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

b)

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	**0**	1	2	3	4	5
∞	∞	∞	∞	1	2	3	**0**	1	2
∞	∞	∞	∞	2	3	4	**0**	1	2
∞	**0**	1	2	3	4	5	1	2	3
∞	1	2	3	4	**0**	1	2	3	4
∞	2	3	4	5	1	2	**0**	1	2
∞	3	4	5	6	2	3	1	2	3
∞	4	5	6	7	3	4	2	3	4

c)

5	4	3	2	1	2	3	2	3	4
4	3	2	1	**0**	1	2	1	2	3
3	2	3	2	1	2	1	**0**	1	2
2	1	2	3	2	2	1	**0**	1	2
1	**0**	1	2	2	3	2	1	2	3
2	1	2	2	1	**0**	1	1	3	3
3	2	3	3	2	1	1	**0**	1	2
4	3	4	4	3	2	2	1	2	3
5	4	5	5	4	3	3	2	3	4

Figure 12: Computation of sequential city-block distances: a) initialisation with zero distances at object-pixel positions; b) calculation of distances from the top-left down to the bottom-right; c) final result

the same comparison is done with the position above:

$$\text{forward scan for all}(i,j) \begin{cases} \text{if} \quad d(i,j) > d(i,j-1)+1 & d(i,j) := d(i,j-1)+1 \\ \text{if} \quad d(i,j) > d(i-1,j)+1 & d(i,j) := d(i-1,j)+1 \end{cases} .$$

This procedure propagates the distances to the right and to the bottom as shown in **Figure 12** b). This direction of propagation has to be reversed in a second scan from the bottom-right up to the top-left position:

$$\text{backward scan for all}(i,j) \begin{cases} \text{if} \quad d(i,j) > d(i,j+1)+1 & d(i,j) := d(i,j+1)+1 \\ \text{if} \quad d(i,j) > d(i+1,j)+1 & d(i,j) := d(i+1,j)+1 \end{cases} ,$$

and the final result in **Figure 12** c) is obtained. The corresponding source code is shown in **Listing 2** (see appendix). In total, the image is scanned twice and two comparisons per scan have to be made at each pixel position.

An alternative approach is based on four scans, but with only one comparison per pixel access. It first processes all columns downward and upward:

$$\begin{array}{rl} \text{downward scan in each column } j: & \text{if} \quad d(i,j) > d(i-1,j)+1 \quad d(i,j) := d(i-1,j)+1 \\ \text{upward scan in each column } j: & \text{if} \quad d(i,j) > d(i+1,j)+1 \quad d(i,j) := d(i+1,j)+1 \end{array} .$$

Since the comparisons are independent of each other, all columns could be processed in parallel. Afterwards, all rows are processed in the same manner:

$$\begin{array}{rl} \text{forward scan in each row } i: & \text{if} \quad d(i,j) > d(i,j-1)+1 \quad d(i,j) := d(i,j-1)+1 \\ \text{backward scan in each row } i: & \text{if} \quad d(i,j) > d(i,j+1)+1 \quad d(i,j) := d(i,j+1)+1 \end{array} .$$

Listing 3 contains the corresponding source code and the example results are shown in **Figure 13**. The final result is exactly the same as in Figure 12 c).

Figure 14 a) depicts the distance transform result for a larger example with ten isolated object pixels. The brighter a pixel is in this picture, the higher is its distance to the closest object pixel. The typical diamond shape of the city-block distance propagation can be seen clearly (compare also Figure 9 b).

a)

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	0	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	0	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	0	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

b)

∞	4	∞	∞	1	5	∞	2	∞	∞
∞	3	∞	∞	**0**	4	∞	1	∞	∞
∞	2	∞	∞	1	3	∞	**0**	∞	∞
∞	1	∞	∞	2	2	∞	**0**	∞	∞
∞	**0**	∞	∞	3	1	∞	1	∞	∞
∞	1	∞	∞	4	**0**	∞	1	∞	∞
∞	2	∞	∞	5	1	∞	**0**	∞	∞
∞	3	∞	∞	6	2	∞	1	∞	∞
∞	4	∞	∞	7	3	∞	2	∞	∞

c)

5	4	3	2	1	2	3	2	3	4
4	3	2	1	**0**	1	2	1	2	3
3	2	3	2	1	2	1	**0**	1	2
2	1	2	3	2	2	1	**0**	1	2
1	**0**	1	2	2	3	2	1	2	3
2	1	2	2	1	**0**	1	1	3	3
3	2	3	3	2	1	1	**0**	1	2
4	3	4	4	3	2	2	1	2	3
5	4	5	5	4	3	3	2	3	4

Figure 13: Parallel computation of city-block distances: a) initialisation with zero distances at object-pixel positions; b) calculation of vertical distances; c) final result

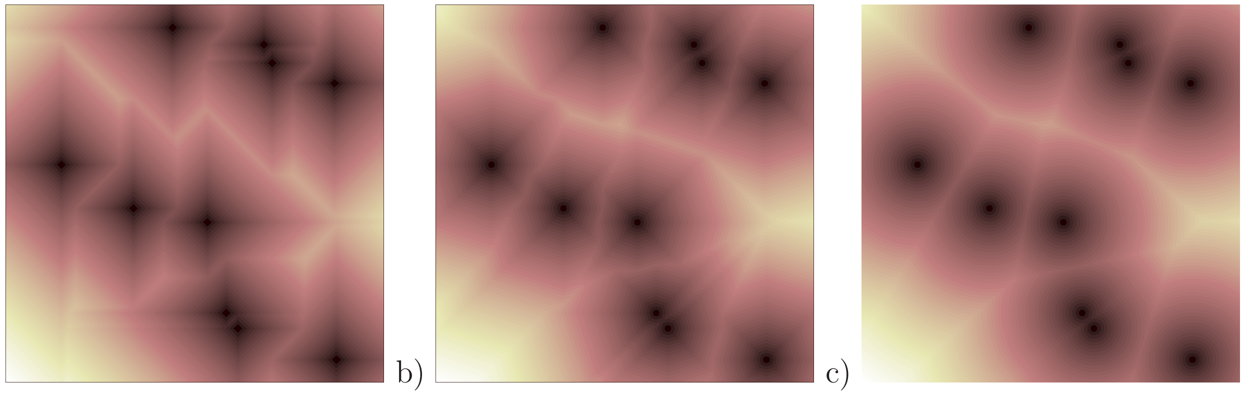


Figure 14: Example of distance transforms based on: a) city-block distance; b) chamfer 4-3 distance; c) propagation of relative positions. The brighter the pixels are, the longer is the distance to the closest object point.

2.2 Chamfer distance transform

In contrast to the city-block distance, the chamfer distance requires considerations in the 8-neighbourhood. Similar to what has been explained in Subsection 2.1, also the chamfer distance can be computed in a sequential manner. Instead of two comparisons per scan and pixel, now four comparisons are needed. The scan from top-left to bottom right makes following decisions:

$$\text{for all}(i,j)\left\{\begin{array}{lll} \text{if} & d(i,j) > d(i,j-1)+3 & d(i,j) := d(i,j-1)+3 \\ \text{if} & d(i,j) > d(i-1,j)+3 & d(i,j) := d(i-1,j)+3 \\ \text{if} & d(i,j) > d(i-1,j-1)+4 & d(i,j) := d(i-1,j-1)+4 \\ \text{if} & d(i,j) > d(i-1,j+1)+4 & d(i,j) := d(i-1,j+1)+4 \end{array}\right. .$$

The backward scan is accordingly

$$\text{for all}(i,j)\left\{\begin{array}{lll} \text{if} & d(i,j) > d(i,j+1)+3 & d(i,j) := d(i,j+1)+3 \\ \text{if} & d(i,j) > d(i+1,j)+3 & d(i,j) := d(i+1,j)+3 \\ \text{if} & d(i,j) > d(i+1,j-1)+4 & d(i,j) := d(i+1,j-1)+4 \\ \text{if} & d(i,j) > d(i+1,j+1)+4 & d(i,j) := d(i+1,j+1)+4 \end{array}\right. .$$

Figure 15 a)+b) depict the intermediate and the final result. The third matrix in figure 15 c) also contains the final result including a normalization with three and subsequent squaring.

a)

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	**0**	3	6	9	12	15
∞	∞	∞	4	3	4	7	**0**	3	6
∞	∞	8	7	6	7	4	**0**	3	6
∞	**0**	3	6	9	8	4	3	4	7
4	3	4	7	10	**0**	3	6	7	8
7	6	7	8	4	3	4	**0**	3	6
10	9	10	8	7	6	4	3	4	7
13	14	12	11	10	8	7	6	7	8

b)

13	10	7	4	3	4	7	6	7	8
10	9	6	3	**0**	3	4	3	4	7
7	6	7	4	3	4	3	**0**	3	6
4	3	4	7	6	6	3	**0**	3	6
3	**0**	3	6	4	3	4	3	4	7
4	3	4	6	3	**0**	3	3	4	7
7	6	7	7	4	3	3	**0**	3	6
10	9	10	8	7	6	4	3	4	7
13	12	12	11	10	8	7	6	7	8

c)

18.8	11.1	5.4	1.8	1	1.8	5.4	4	5.4	7.1
11.1	9	4	1	**0**	1	1.8	1	1.8	5.4
5.4	4	5.4	1.8	1	1.8	1	**0**	1	4
1.8	1	1.8	5.4	4	4	1	**0**	1	4
1.0	**0**	1	4	1.8	1	1.8	1	1.8	5.4
1.8	1	1.8	4	1	**0**	1	1	1.8	5.4
5.4	4	5.4	5.4	1.8	1	1	**0**	1	4
11.1	9	11.1	7.1	5.4	4	1.8	1	1.8	5.4
18.8	16	16	13.4	11.1	7.1	5.4	4	5.4	7.1

Figure 15: Computation of chamfer 4-3 distances: a) propagation of distances from the top-left down to the bottom-right; b) final result $d_{i,j}$ after backward propagation; c) rounded values of $(d_{i,j}/3)^2$, integer numbers are exact values

a)

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	0,0	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	0,0	∞	∞
∞	∞	∞	∞	∞	∞	∞	0,0	∞	∞
∞	0,0	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	0,0	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	0,0	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

b)

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
0,4	0,3	0,2	0,1	**0,0**	0,1	0,2	0,3	0,4	0,5
1,4	1,3	1,2	1,1	1,0	1,1	0,1	**0,0**	0,1	0,2
2,4	2,3	2,2	2,1	2,0	0,2	0,1	**0,0**	0,1	0,2
0,1	**0,0**	0,1	0,2	3,0	1,2	1,1	1,0	1,1	1,2
1,1	1,0	1,1	0,2	0,1	**0,0**	0,1	2,0	2,1	2,2
2,1	2,0	2,1	1,2	1,1	1,0	0,1	**0,0**	0,1	0,2
3,1	3,0	3,1	2,2	2,1	2,0	1,1	1,0	1,1	1,2
4,1	4,0	4,1	3,2	3,1	2,2	2,1	2,0	2,1	2,2

c)

4,1	1,3	1,2	1,1	1,0	1,1	2,1	2,0	2,1	2,2
3,1	0,3	0,2	0,1	**0,0**	0,1	1,1	1,0	1,1	1,2
2,1	2,0	1,1	1,1	1,0	1,1	0,1	**0,0**	0,1	0,2
1,1	1,0	1,1	2,1	2,0	0,2	0,1	**0,0**	0,1	0,2
0,1	**0,0**	0,1	0,2	1,1	1,0	1,1	1,0	1,1	1,2
1,1	1,0	1,1	0,2	0,1	**0,0**	0,1	1,0	1,1	1,2
2,1	2,0	2,1	1,2	1,1	1,0	0,1	**0,0**	0,1	0,2
3,1	3,0	3,1	2,2	2,1	2,0	1,1	1,0	1,1	1,2
4,1	4,0	4,1	3,2	3,1	2,2	2,1	2,0	2,1	2,2

Figure 16: Approximate Euclidean distance transform: a) initialisation of object pixel positions; b) propagated relative positions (Δ_y, Δ_x) after first scan from top to bottom; c) final result (Δ_y, Δ_x) after propagation from bottom to top;

This modification scales the values back to a horizontal distance equal to one between two pixels and allows the comparison with exact Euclidean distances in Figure 18 c).

The shape of the distance propagation using chamfer distances has changed to an octagon, see **Figure 14** b) .

2.3 Approximate Euclidean Distance Transform

Danielsson had the idea to not propagate the distances but the relative vertical and horizontal positions [Dan80]. Object pixels always have a relative position of $[\Delta_y, \Delta_x] = [0, 0]$, see **Figure 16** a). Each horizontal step increments the horizontal relative position Δ_x and each vertical step increments the vertical relative position Δ_y. A diagonal step increments both. The Euclidean distance can be computed with $d = (\Delta_y^2 + \Delta_x^2)^{0.5}$.

The source code in **Listing 4**, **Listing 5**, and **Listing 6** differs somewhat from Danielsson's proposal and follows the modifications of [Gre07].

In addition to the distance matrix `distMat` that has to be generated, two other matrices `distMatY` and `distMatX` have to be maintained. They store the propagated relative positions. Based on these relative positions, the squared Euclidean distances have to be determined. Instead of computing these distances again and again, a function `getDist()` is used. It calculates the different distances only once, stores these values in a matrix `preCalc`, and also utilizes the symmetric property of the metric.

The source code in Listings 5 and 6 is somewhat lengthy, since several decisions have to be made. The processing comprises two scans: the first is starting in the second row going down to the bottom row and the second is operating in opposite direction. In each scan, the rows are processed forward and backward. The lines 3-14 in Listing 5, for example, perform a vertical propagation of relative positions. The values from the top neighbour position are taken (`yr = distMatY(i-1,j); xr = distMatX(i-1,j);`). The vertical component is incremented (`yr+1`) and the corresponding squared distance is determined using the function `getDist()`. If the current distance is longer than this new one (`if distMat(i,j) > dist`), it is replaced and also the relative positions are updated (`distMat(i,j) = dist; distMatY(i,j) = yr+1; distMatY(i,j) = xr`). However, function `getDist()` may not be accessed when the relative positions are larger than the dimensions of matrix `preCalc` (`if yr < maxDist`).

This procedure is repeated for the two other positions in the causal neighbourhood `(i,j-1)` and `(i-1,j-1)`. The subsequent backward scan for this row checks the positions `(i,j+1)` and `(i-1,j+1)`. **Figure 16** b) shows the result of this first scan. Note that the sorted pairs a, b and b, a correspond to the same distance $d^2 = a^2 + b^2$. However, the derived order of a and b indicates to which object pixel each background pixel has been assigned.

The second scan propagates the minimum distances from the bottom row upwards (Listing 6) in a very similar manner and produces the final result (**Figure 16** c). For the given example, this corresponds to the exact Euclidean distances as shown in Figure 18.

Under special circumstances, it is impossible to derive the correct distance. This is because the algorithm depends on the propagation of distances, while the assignment of the closest object pixel can lead to disconnected background pixels. This has already been mentioned and explained by Danielsson himself [Dan80] and different numeric examples have been given in [Bai05] and [Fab08]. One example can be seen in **Figure 17**. The continuous space is separated in regions by drawing the perpendicular bisector between two points. There is a clear assignment of each position to one of the three point, Figure 17 a). In discrete space, the three object pixels (marked with a cross) also split the set of background pixels in three region, Figure 17 b). The white lines show the separation in continuous space. The discretization causes a disruption of the middle region. The approach of Danielsson is not able to take account of this disruption and assigns the isolated pixel to the bottom region with a relative position of $1, 13$. There is no chance to propagate the positions from ‘4,10’ to the correct relative distance of ‘5,12’ via ‘5,11’ or via ‘4,11’ because the corresponding positions are closer to the other object pixels.

A typical visual appearance of this distance transform result is shown in **Figure 14** c). It looks in principle like a result produced by an exact Euclidean distance transform.

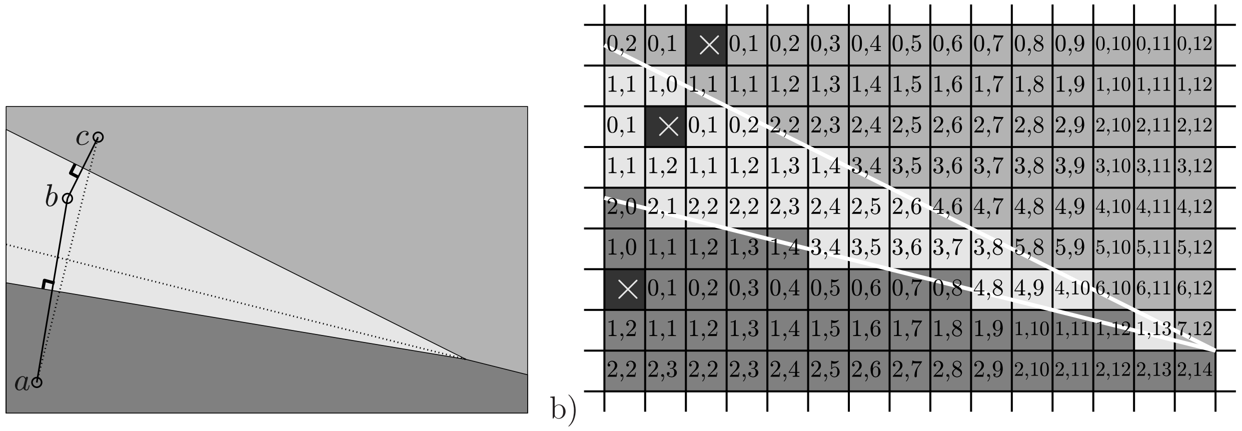
a) b)


Figure 17: Assignment of points: a) continuous space separated in three regions which are closest to either point a, b, or c; b) example result of Danielssons approach. Object pixels are marked with $\times$. The bright isolated pixel has been assigned to the wrong object pixel with a distance of $d^2 = 1^2 + 13^2 = 170$. However, its correct distance is $d^2 = 5^2 + 12^2 = 169$ with relation to the object pixel in the middle.

3 Exact Euclidean Distance Transform

Some applications depend on the exact calculation of the distances between each background pixel and the nearest object pixel. Thus, the following explanations refer to the case shown in Figure 1 c). In principle, one could first identify the inner contours of all objects. The inner contour includes all object pixels that have at least one background pixel in their neighbourhood. In a second step, one could compute the Euclidean distances from each background pixel to all contour pixels and find the minimum. This minimum distance is then stored at the position of the examined background pixel.

In worst case, each object pixel could be a contour pixel[1]. Let n_O denote the number of contour pixels. The total number of pixels is given by the image size $L \times M$. The number of background pixels is then $n_B = L \cdot M - n_O$. The number n_c of distance calculations would be

$$n_c = n_B \cdot n_O = (L \cdot M - n_O) \cdot n_O \tag{3}$$

The maximum of required calculations $\max(n_c) = (L{\cdot}M)^2/4$ is reached for $n_B = n_O = (L{\cdot}M)/2$. Consequently, this naive approach has a quadratic complexity with respect to the image size: $O((LM)^2)$.

For a better understanding of the following text, the most important variables are summarized in **Table 1**.

3.1 Basic approach

Many researchers have thought of algorithms that can compute the exact Euclidean distance transform (EEDT) in a faster manner. Some of these successful methods are based on the separate processing of all dimensions [Mei00, Mau03, Fel12]. Let us take an example with six object pixels. The distance matrix in **Figure 18** a) shows the positions of object pixels with a value of zero. The following derivations use squared Euclidean distances

[1]For example, each object could be represented by a single isolated pixel.

Table 1: Notation

d	...	distance value
D	...	squared distance $D = d^2$
Δ	...	difference between elements of coordinates
i	...	row index
i_0	...	index of current or particular row
i_n	...	vertical coordinate of an object pixels $\mathbf{p}_n$
j	...	column index
j_0	...	index of current or particular column
j_n	...	horizontal coordinate of an object pixels $\mathbf{p}_n$
j_s	...	horizontal location of an intersection, $j_s \in \mathbb{R}$
k	...	column index related to object pixels
L	...	number of columns
M	...	number of rows
n	...	index for object pixels
$\mathbf{p}_n$	...	object pixel

a)

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	0	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	0	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	0	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	0	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

b)

∞	16	∞	∞	1	25	∞	4	∞	∞
∞	9	∞	∞	**0**	16	∞	1	∞	∞
∞	4	∞	∞	1	9	∞	**0**	∞	∞
∞	1	∞	∞	4	4	∞	**0**	∞	∞
∞	**0**	∞	∞	9	1	∞	1	∞	∞
∞	1	∞	∞	16	**0**	∞	1	∞	∞
∞	4	∞	∞	25	1	∞	**0**	∞	∞
∞	9	∞	∞	36	4	∞	1	∞	∞
∞	16	∞	∞	49	9	∞	4	∞	∞

c)

17	10	5	2	1	2	5	4	5	8
10	9	4	1	**0**	1	2	1	2	5
5	4	5	2	1	2	1	**0**	1	4
2	1	2	5	4	4	1	**0**	1	4
1	**0**	1	4	2	1	2	1	2	5
2	1	2	4	1	**0**	1	1	2	5
5	4	5	5	2	1	1	**0**	1	4
10	9	10	8	5	4	2	1	2	5
17	16	17	13	10	8	5	4	5	8

Figure 18: Computation of squared Euclidean distances: a) initialisation with zero distances at object-pixel positions; b) calculation of vertical distances; c) final result

$$D = d^2 = (p_x - q_x)^2 + (p_y - q_y)^2 \ , \tag{4}$$

with $\mathbf{p} = (p_x, p_y)$ being the position of a background pixel and $\mathbf{q} = (q_x, q_y)$ being the position of an object pixel. The square root can be drawn (if necessary) after all squared distances have been determined.

The distances for the background pixels are not known yet and they are initialised with a sufficiently large value higher than $(M-1)^2 + (L-1)^2$, which represents the maximum possible distance for a $M \times L$ matrix. In Figure 18, this value is indicated by '∞'.

From (4) it can be seen that the distance consists of two additive terms, which can be computed independently. For example, one could first calculate all vertical components $D_\mathrm{v} = (p_y - q_y)^2$. The result is depicted in **Figure 18** b).

In a second step, the horizontal components $D_\mathrm{h} = (p_x - q_x)^2$ have to be determined row by row. At all positions in each row, it has to be checked which combination $D = D_\mathrm{v} + D_\mathrm{h}$ of vertical and horizontal components results in the shortest distance. Starting in the top-left corner, there is no distance assigned yet. The row is scanned to the right for the next available distance component. A value of $D_\mathrm{v} = 16$ can be found at a horizontal distance of $d_\mathrm{h} = 1$. The combination of both components is $D_2 = 1^2 + 16 = 17$. The index k in D_k indicates the column of the used vertical distance. This is the first candidate. The second can be computed at a horizontal distance of 4: $D_5 = 4^2 + 1 = 17$. Two more candidates are available: $D_6 = 5^2 + 25 = 50$ and $D_8 = 7^2 + 4 = 53$. The best candidate is obviously

$$D^* = \min_i(D_i) = 17 \ .$$

This value is written to the current pixel position. The candidates for the second pixel in the first row are:

$$\begin{aligned}
D_2 &= 0^2 + 16 = 16 \quad \text{(pure vertical distance)}\\
D_5 &= 3^2 + \ \ 1 = 10\\
D_6 &= 4^2 + 25 = 41\\
D_8 &= 6^2 + \ \ 4 = 40 \ .
\end{aligned}$$

The already existing value of 16 has to be replaced by $D^* = 10$, because this is the shortest distance to any object pixel. This procedure continues until all distances are fixed for this row. Then all other rows are processed in the same manner.

Listing 7 contains a possible algorithm for this procedure. This source code allows an estimation of the corresponding complexity. Lines 15 to 34 compute the vertical components. Interestingly, there is no multiplication involved despite the fact that $(p_y - q_y) \cdot (p_y - q_y)$ has to be computed for each background pixel. The outer loop `for j=1:L` steps through all columns. There are two inner loops: `for i=2:M` and `for i=M-1:-1:1`. The first one scans the current column downwards starting with the second row and the second loop scans the same column upwards. Both try to propagate distances in such a manner that larger distance entries are overwritten by possibly smaller ones.

With respect to the example in Figure 18 a), the downward scan firstly enfolds its impact in the second column (counted from the left) at the position right under the object pixel with distance equal to zero. At this position it finds a distance value that is larger ($= \infty$) than the distance above plus the actual distance step, which had been initialized with `distStep=1`. The current value is set to $0 + 1 = 1$ and the distance step is increased by two. What's the deal with that increment?

Given two consecutive numbers x and $x + 1$, the difference between their squared values is $(x + 1)^2 - x^2 = 2x + 1$. Starting with $x = 0$, the first difference is equal to one (`distStep=1`). Then the difference between the squared values grows by 2 (`distStep = distStep + 2;`) with each increment of x. Using this step as an additive term in `distMat(i,j) = distMat(i-1,j) + distStep`, the algorithm computes the required squared distance values without any multiplication.

As soon as the position of an object pixel is reached, the current distance value is not larger than the propagated one; the distance value remains unchanged and `distStep` is reset to one. The result of the downwards propagation is shown in **Figure 19**. The upward scan within a column does not change anything before the first object pixel is passed. After this position the distances are update if they are larger than the propagated values, see 18 b). These scans are rather fast and there is only opportunity for improvements in the upward scan, if there are many columns without any object pixel (entire column could be skipped in the upward scan) or

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	**0**	∞	∞	∞	∞	∞
∞	∞	∞	∞	1	∞	∞	**0**	∞	∞
∞	∞	∞	∞	4	∞	∞	**0**	∞	∞
∞	**0**	∞	∞	9	∞	∞	1	∞	∞
∞	1	∞	∞	16	**0**	∞	4	∞	∞
∞	4	∞	∞	25	1	∞	**0**	∞	∞
∞	9	∞	∞	36	4	∞	1	∞	∞
∞	16	∞	∞	49	9	∞	4	∞	∞

Figure 19: Result after downward propagation of squared Euclidean distances, see example from Figure 18 a)

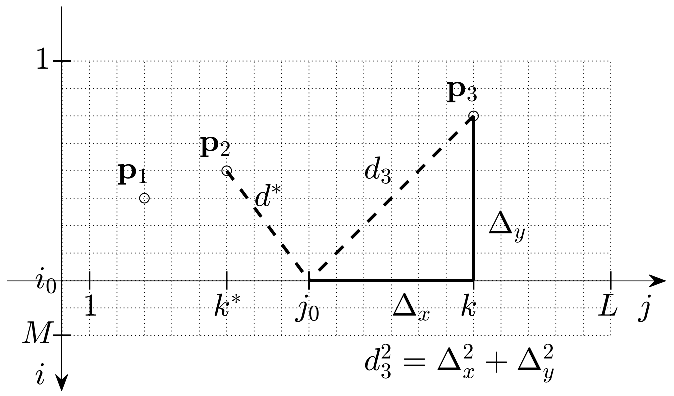


Figure 20: Looking for closest object point p_k for current background pixel at position (i_0, j_0). The current minimum distance is D^* related to point $\mathbf{p}_2$ from column k^*. The distance measure d_3 cannot be shorter because its horizontal component Δ_x is already longer than the current minimum: $\Delta_x > d^*$.

the object pixels concentrate in the upper rows of the image (lower part of the column could be skipped). In maximum there are $L \cdot M$ distances to be computed. So, the vertical propagation has linear complexity with respect to the image size.

The processing of rows needs more efforts and defines the complexity of the entire approach, Listing 7 lines 37-52. The outer loop `for i=1:M` takes care of all matrix rows. The corresponding row of vertical distances is copied to a new vector `distMatV` because the values in matrix `distMat` can be overwritten. In lines 40 and 43, there are two nested inner loops, both scanning all positions within the current row. Index j of the first inner loop defines the position for which the minimum distance has to be found. The second inner loop scans for all possible candidates of vertical distances `distMatV(k)`.

Figure 20 shows a situation where a background pixel at position (i_0, j_0) has to be assigned the distance to the closest object pixel $\mathbf{p_k}$. Scanning the line i_0 from $k = 1$ towards $k = L$, the current point has already been compared to the points p_1 and p_2 resulting to a minimum distance of $d^* = \sqrt{4^2 + 3^2} = 5$ in this example. The next vertical distance $D_\mathrm{v} < \infty$ can be found in the column of point $\mathbf{p_3}$.

In the worst case, all positions j_0 have to be compared with all positions k with $1 \leq j_0, k \leq L$, i.e. $L \cdot (L-1)$ distances have to be computed. Taking all rows into account, we have $M \cdot L \cdot (L-1)$ distances to calculate leading to a $O(ML^2)$ complexity, which is at least somewhat lower than for the naive approach mentioned in the beginning of Section 3.

3.2 Faster approaches

The most important question is, whether it is possible to skip some of the distance calculations. While the determination of vertical distances is already quite fast and has only linear $O(M)$ complexity, the calculation of distances along the rows has some room for optimizations. With respect to the basic approach discussed above, there are some tricks to speed up the processing.

Change order of column and row processing

Depending on the aspect ratio of the given image, the complexity $O(LM^2)$ is lower than $O(ML^2)$ if $M < L$. If the complexity of the row-wise processing will not reduced by another mechanism, the order of processing columns and rows should be exchanged in this case.

Avoid unnecessary computations

The computations in lines 43-49 of Listing 7 should only be executed if there is a chance to find a new minimum distance. Columns without any object point contain infinite (`=maxDist`) vertical distances. So, checking '`if distMatV(k) < maxDist`', it can be ensured that variable k is not pointing at such column and unnecessary computations can be avoided. However, if most of the columns are occupied by at least one object pixel, this has little effect.

A second simple trick that can be combined with the first one concerns the horizontal component of the distances to be computed.

As soon as a position $k = j_0 + \Delta_x$ with $\Delta_x > d^*$ is reached, no shorter path than d^* can be found and thus the scan can be terminated. This situation is shown in Figure 20. The current minimum distance d^* refers to point $\mathbf{p_2}$ and the next available point $\mathbf{p_3}$ has a horizontal distance that already is larger than d^* and the processing of loop `for k=1:L` can be stopped. This leads to large saving effects when j_0 is small. However, the effect decreases the more the current position j_0 moves towards L. If the scanning process is split into a forward scan $j_0 < k \leq L$, which can be processed as described and a backward scan from $k = j_0 - 1$ to $k = 1$, then it is possible to stop both scanning processes as soon as $|k - j_0| \geq d^*$ is reached. These modifications are implemented in **Listing 8**. Remember that all distances are squared ones.

3.3 Fast exact Euclidean distance transforms

The previous Subsections explained how the exact Euclidean distance transform can be realised and they proposed to skip unnecessary computations. However, the discussed methods for doing so are quite simple and have impact only under special conditions. As mentioned in Subsection 3.1, only the vertical distance component D_v can be determined in linear time ($O(M)$) while the horizontal distance component D_h still needs $O(L^2)$ leading to a total complexity of $O(ML^2)$. However, based on deeper considerations, it is possible to reduce the complexity to $O(ML)$ as different scientists have proven. With other words: also the determination of horizontal distances can be performed in linear time.

The explanations in Subsection 3.1 regarding the combination of the vertical and the horizontal component to find the final distance in (4) can be viewed from another perspective. Each object point $\mathbf{p}_n$ located at (i_n, j_n) contributes to row i_0 with a vertical component

j, k	1	2	3	4	5	6	7	8	9	10
$D_{k,\mathrm{v}}$	∞	**16**	∞	∞	**1**	**25**	∞	**4**	∞	∞
D_2	17	**16**	17	20	25	32	41	52	65	80
D_5	17	10	5	2	**1**	2	5	10	17	26
D_6	50	41	34	29	26	**25**	26	29	34	41
D_8	53	40	29	20	13	8	5	**4**	5	8
min	17	10	5	2	1	2	5	4	5	8
k^*	2/5	5	5	5	5	5	5/8	8	8	8

Figure 21: Propagation of horizontal distances starting from the vertical components $D_{k,\mathrm{v}}$ taken from the first row of the matrix in Figure 18 b). Variable k^* indicates which column contributes the smallest distance D.

$D_{n,\mathrm{v}}(i_0) = (i_n - i_0)^2$. According to Figure 20, its horizontal component is $D_{n,\mathrm{h}}(j_0) = (j_n - j_0)^2$. For each point $\mathbf{p}_n$ and a given row i_0, all distances $D_n(i_0, j_0) = D_{n,\mathrm{v}}(i_0) + D_{n,\mathrm{h}}(j_0)$ for all horizontal positions j_0, with $1 \leq j_0 \leq L$ could be computed. With respect to the example shown in Figure 18 b), we could take the first row containing four vertical components $\{D_{n,\mathrm{v}}(i_0 = 1)\} = \{16, 1, 25, 4\}$ and could complement them separately with all possible horizontal components $D_{n,\mathrm{h}}(j_0)$. The result is shown in **Figure 21**.

Since we are looking for the minimum distance, the minimum value is determined for each column j. This results to a row of distances which is equal to what we already got in Figure 18 c), in first row.

Additionally, we can keep track of which column k contributes the minimum distance for a certain column j_0. As can be seen in Figure 21, the background pixel at position $j = 1$ has a minimum distance to the object point lying in column $k = 2$ and the same distance to the point lying in column $k = 5$. The background points in columns $2 \leq j \leq 6$ are closest to the object point from column $k = 5$. For $j = 7$, again two object points share by chance the same distance to the background point a.s.o. The object pixel from column $k = 6$ is not taken into account for this row because already its vertical component $D_{6,\mathrm{v}} = 25$ is to high.

Method by Meijster, Roerdink, and Hesselink

Looking at Figure 21, it becomes clear that the values in the rows for D_k follow a quadratic equation

$$D_k = D_{k,\mathrm{v}} + (j - k)^2 \tag{5}$$

with j being the independent variable. Hence, the distance $D_k = f(j)$ can be visualized as parabolas, see **Figure 22**. When looking for minimal distances $D(i_0, j_0) = \min_k(D_k)$, it becomes clear that only the lower envelop of all parabolas is of interest and object points which do not contribute to this envelop can be ignored in the distance computations.

It also can deduced from Figure 22 that the intersection points of contributing parabolas define the range in which a parabola (i.e. the according object point) is the closest one. This

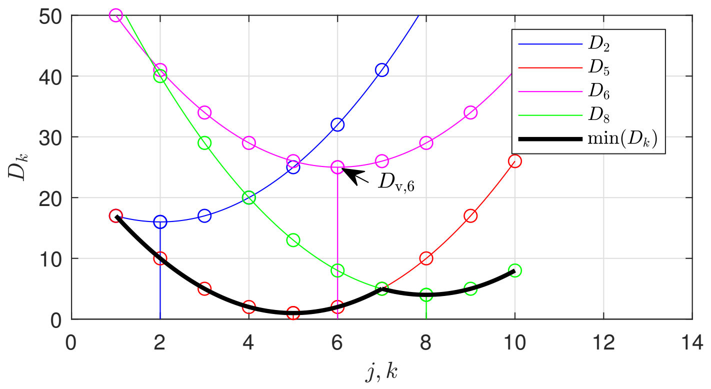


Figure 22: Graphical visualization of distances D_k from Figure 21. The vertical distances $D_{k,\mathrm{v}}$ are located at the parabola vertices. Only $\min_k(D_k)$ (lower envelop) is of interest.

idea has been proposed by Meijster et al [Mei00]. Working source code has been provided by Felsenszwalb and Hutterlocher [Fel12] while it is unclear whether they had been aware of Meijsters work.

So, the key point is to determine the position of the intersection of two parabolas. Let us assume that there are two parabolas with vertices at the columns k and l. Since they have a single intersection, the position j_s of this intersection can be determined using (5) with following equation:

$$\begin{aligned}
D_{k,\mathrm{v}} + (j_s - k)^2 &= D_{l,\mathrm{v}} + (j_s - l)^2 \\
(j_s - k)^2 - (j_s - l)^2 &= D_{l,\mathrm{v}} - D_{k,\mathrm{v}} \\
j_s^2 - 2kj_s + k^2 - j_s^2 + 2lj_s - l^2 &= D_{l,\mathrm{v}} - D_{k,\mathrm{v}} \\
2lj_s - 2kj_s &= D_{l,\mathrm{v}} - D_{k,\mathrm{v}} - k^2 + l^2 \\
j_s \cdot 2(l - k) &= D_{l,\mathrm{v}} - D_{k,\mathrm{v}} - k^2 + l^2 \\
j_s &= \frac{D_{l,\mathrm{v}} - D_{k,\mathrm{v}} - k^2 + l^2}{2(l - k)} \qquad (6)
\end{aligned}$$

Using the example in Figure 22, an algorithm is explained in the following, which can significantly reduce the complexity. Two vectors are needed for storing the required information of the lower envelop. The first vector ks will contain the positions $k \in \{j_n\}$ of all parabolas which finally contribute to the lower envelop and the second vector js will contain the positions j from which a parabola starts to contribute to the lower envelop. Theses positions can be derived from the intersections j_s rounded to the next larger integer value.

The vectors have to be initialized with the parameters of the first, possibly virtual, parabola: $ks(1) = 1$ and $js(1) = -\infty$. The huge negative value serves as stopping criterion for the algorithm as will be explained below. The corresponding source code is shown in **Listing 9** in lines 6 - 12.

The position of the first available (blue) parabola is $k = 2$ and the intersection j_s with the virtual parabola is per definition larger than $-\infty$. The information of the new parabola is stored: $ks(2) = 2$ and $js(2) = \max(1, j_s)$. It has to be ensured that the lower border starts not before $j = 1$.

The position of the next available (red) parabola is $k = 5$ and the intersection with the previous is exactly at position $j_s = 1$. Since $j_s <= js(2)$ holds, the blue parabola is hidden by the new one and does not contribute anymore. Its parameters can be overwritten: $ks(2) = 5$, $js(2) = 1$. This check has now to be done with all previous parabolas (lines 21-26). As soon as the intersection is $j_s > js(idx)$, the previous parabola contributes to the envelop and the information of the current parabola is appended to the vectors ks and js. The third (magenta) parabola of the example in Figure 22 intersect with the previously stored (red) parabola at a position beyond L and thus can be ignored. The fourth and last (green) parabola has an intersection with the red one at $j_s = 7$. This is larger than the previously stored value of $js(2) = 1$, so it does not hide the previous parabola and the vectors must be complemented with the parameters of the green parabola: $ks(3) = 8$, $js(3) = 7$ (lines 27 - 31). There is no further parabola available and the intersections vector can be closed with $js(1) = 1$ and $js(idx + 1) = L + 1$ defining the maximum range. The vectors now read as $js = (1, 1, 7, 11)$ and $ks = (1, 5, 8)$ giving the information that parabola at $k = 1$ is either not existing or not contributing ($1 \leq j < 1$!), parabola $k = 5$ defines the distances for the positions $1 \leq j < 7$ and parabola $k = 8$ is responsible for the distances at $7 \leq j < 11$. Using equation (5), the squared Euclidean distance can be calculated for all positions j (lines 37-42). This procedure is performed for each row of the distance matrix.

In total, each row is scanned twice. The first scan identifies the contributing object points (parabolas) and the second scan applies the collected information and calculates the final distances $D(i : 0, j_0)$. Even though, the first scan uses a double while-loop, its complexity can be considered as being linear ($O(L)$). That means, the entire method has $O(M \cdot L)$ complexity.

4 Related things

The method by Meijster *et al* is not the only possibility for the efficient determination of relevant object points. Maurer *et al* proposed a technique that is based on the computation of lines separating the regions which belong to different object pixels [Mau03], see Figure 17. This method also efficiently identifies the contributing object points for each row and reaches $O(M \cdot L)$ complexity.

A Source Code

Listing 1: Examples for distance calculations

```
N = 6;
disp('Euclidean') % Euclidean distances
A = ((0:N-1).^2 )' * ones(1, N);
B = A';
E = A+B % squared Euclidean distances

disp('Chamfer') % Chamfer distances
C = zeros(N);
diagStep = sqrt(2);
%diagStep = 4./3;
for i = 1:N
  for j = 1:N
    C(i,j) = j-1 + i-1;
    if i > 1 && j > 1
      C(i,j) = C(i-1,j-1) + diagStep;
    end
  end
end
Cr = round( 10* C.^2) / 10 % rounded squared
     Chamfer distances
D = sqrt(E) - sqrt(Cr) % difference between
     Euclidean and Chamfer

sum( abs(D(:))) / (N*N) % average error
max(abs(D(:))) % max absolute error
```

Listing 2: Example of sequential city-block distance transformation

```
L = 10; %number of columns
M = 9; % number of rows
maxDist = L + M; % sufficient large distance
distMat = ones(M,L) * maxDist; % initilize

% example coordinates of object pixels
x = [ 2, 5, 6, 8, 8, 8]; % horizontal position
y = [ 5, 2, 6, 3, 4, 7]; % vertical position

for n = 1: length (x)
  distMat(y(n), x(n)) = 0; % set object pixels
end

% scan from top-left to bottom-right
for i = 2:M % leave top border
  for j = 2:L % leave left border
    if distMat(i,j) > distMat( i-1, j) + 1
       distMat(i,j) = distMat( i-1, j) + 1;
    end
    if distMat(i,j) > distMat( i, j-1) + 1
       distMat(i,j) = distMat( i, j-1) + 1;
    end
  end
end

% scan bottom line from right to left
for j = L-1:-1:1 % ignore left border
    if distMat(M,j) > distMat( M, j+1) + 1
      distMat(M,j) = distMat( M, j+1) + 1;
    end
end
% scan right column from bottom to top
for i = M-1:-1:1 % ignore left border
    if distMat(i,L) > distMat( i+1, L) + 1
      distMat(i,L) = distMat( i+1, L) + 1;
    end
end
% scan from bottom-right to top-left
for i = M-1:-1:1 % ignore bottom border
  for j = L-1:-1:1 % ignore right border
    % look backwards
    if distMat(i,j) > distMat( i+1, j) + 1
      distMat(i,j) = distMat( i+1, j) + 1;
    end
    if distMat(i,j) > distMat( i, j+1) + 1
      distMat(i,j) = distMat( i, j+1) + 1;
    end
  end
end
distMat % outputs the resulting distances
```

Listing 3: Example of parallel city-block distance transformation

```
L = 10; %number of columns
M = 9; % number of rows
maxDist = L + M; % sufficient large distance
distMat = ones(M,L) * maxDist; % initilize

% coordinates of object pixels
x = [ 2, 5, 6, 8, 8, 8]; % horizontal position
y = [ 5, 2, 6, 3, 4, 7]; % vertical position

for n = 1: length (x)
  distMat(y(n), x(n)) = 0; % set object pixels
end

% scan from top to bottom and back
for j = 1:L % for all columns
  % downward
  for i = 2:M % leave top border
    if distMat(i,j) > distMat( i-1, j) + 1
        distMat(i,j) = distMat( i-1, j) + 1;
    end
  end
  % upward
  for i = M-1:-1:1 % leave bottom border
    if distMat(i,j) > distMat( i+1, j) + 1
        distMat(i,j) = distMat( i+1, j) + 1;
    end
  end
end

% scan from left to right and back
for i = 1:M % for all rows
  for j = 2:L% ignore left border
    % to the right
    if distMat(i,j) > distMat( i, j-1) + 1
       distMat(i,j) = distMat( i, j-1) + 1;
    end
  end
  for j = L-1:-1:1% ignore left border
    % to the left
    if distMat(i,j) > distMat( i, j+1) + 1
       distMat(i,j) = distMat( i, j+1) + 1;
    end
  end
end

distMat % outputs the resulting distances
```

Listing 4: Example of approximate Eucidean distance transformation (part 1)

```
L = 10; %number of columns
M = 9; % number of rows
maxDist = L*L + M*M; % sufficient large distance
distMat = ones(M,L) * maxDist; % distance matrix
 % initilize relative positions
distMatX = ones(M,L) * maxDist;
distMatY = ones(M,L) * maxDist;
% matrix for pre-calculated distances
preCalc = -ones( max(M, L));

% coordinates of object pixels
x = [ 2, 5, 6, 8, 8, 8]; % horizontal position
y = [ 5, 2, 6, 3, 4, 7]; % vertical position

% set object pixels
for n = 1: length (x)
  distMat(y(n), x(n)) = 0;
  distMatY(y(n), x(n)) = 0;
  distMatX(y(n), x(n)) = 0;
end

% first scan
:
% secon scan
:

----------------------------
function [dist] = getDist( dy, dx, preCalc)
  % '+1' because of matlab indexing
  if preCalc(dy+1, dx+1) < 0
      dist = dy * dy + dx * dx;
      preCalc(dy+1, dx+1) = dist;
      preCalc(dx+1, dy+1) = dist; % symmetric
  else % use pre-calculation
    dist = preCalc(dy+1, dx+1);
  end
end
```

Listing 5: Example of approximate Eucidean distance transformation (part 2)

```
% first scan
for i = 2:M % exclude top border
  for j = 1:L %
    yr = distMatY(i-1,j); % top neighbour
    if yr < maxDist % something to propagate
      xr = distMatX(i-1,j);
      dist = getDist( yr+1, xr); % new distance
      if distMat(i,j) > dist
        distMat(i,j) = dist; % replace
        distMatY(i,j) = yr+1;
        distMatX(i,j) = xr;
      end
    end
  end
  for j = 2:L % exclude left border
      yr = distMatY(i,j-1); % left neighbour
      if yr < maxDist
        xr = distMatX(i,j-1);
        dist = getDist( yr, xr+1);
        if distMat(i,j) > dist
          distMat(i,j) = dist;
          distMatY(i,j) = yr;
          distMatX(i,j) = xr+1;
        end
      end
      yr = distMatY(i-1,j-1); % top-left neighbour
      if yr < maxDist
        xr = distMatX(i-1,j-1);
        dist = getDist( yr+1, xr+1);
        if distMat(i,j) > dist
          distMat(i,j) = dist;
          distMatY(i,j) = yr+1;
          distMatX(i,j) = xr+1;
        end
      end
  end
  for j = L-1:-1:1 % backward
      yr = distMatY(i-1,j+1); % top-right neighbour
      if yr < maxDist
        xr = distMatX(i-1,j+1);
        dist = getDist( yr+1, xr+1);
        if distMat(i,j) > dist
          distMat(i,j) = dist;
          distMatY(i,j) = yr+1;
          distMatX(i,j) = xr+1;
        end
      end
      yr = distMatY(i,j+1);
      if yr < maxDist
        xr = distMatX(i,j+1); % right neighbour
        dist = getDist( yr, xr+1);
        if distMat(i,j) > dist
          distMat(i,j) = dist;
          distMatY(i,j) = yr;
          distMatX(i,j) = xr+1;
        end
      end
  end
end
```

Listing 6: Example of approximate Eucidean distance transformation (part 3)

```
% second scan
for i = M-1:-1:1 % ignore bottom border
  for j = 1:L
    yr = distMatY(i+1,j);
    if yr < maxDist
      xr = distMatX(i+1,j);
      dist = getDist( yr+1, xr);
      if distMat(i,j) > dist
        distMat(i,j) = dist;
        distMatY(i,j) = yr+1;
        distMatX(i,j) = xr;
      end
    end
  end
  for j = 2:L % ignore left border
    yr = distMatY(i,j-1);
    if yr < maxDist
      xr = distMatX(i,j-1);
      dist = getDist( yr, xr+1);
      if distMat(i,j) > dist
        distMat(i,j) = dist;
        distMatY(i,j) = yr;
        distMatX(i,j) = xr+1;
      end
    end
    yr = distMatY(i+1,j-1);
    if yr < maxDist
      xr = distMatX(i+1,j-1);
      dist = getDist( yr+1, xr+1);
      if distMat(i,j) > dist
        distMat(i,j) = dist;
        distMatY(i,j) = yr+1;
        distMatX(i,j) = xr+1;
      end
    end
  end
  for j = L-1:-1:1 % ignore right border
    yr = distMatY(i,j+1);
    if yr < maxDist
      xr = distMatX(i,j+1);
      dist = getDist( yr, xr+1);
      if distMat(i,j) > dist
        distMat(i,j) = dist;
        distMatY(i,j) = yr;
        distMatX(i,j) = xr+1;
      end
    end
    yr = distMatY(i+1,j+1);
    if yr < maxDist
      xr = distMatX(i+1,j+1);
      dist = getDist( yr+1, xr+1);
      if distMat(i,j) > dist
        distMat(i,j) = dist;
        distMatY(i,j) = yr+1;
        distMatX(i,j) = xr+1;
      end
    end
  end
end
```

Listing 7: Simple EEDT algorithm

```
L = 10; %number of columns
M = 9; % number of rows
maxDIst = L*L + M*M; % sufficient large distance
distMat = ones(M,L) * maxDist; % initilize distance matrix

% coordinates of object pixels
x = [ 2, 5, 6, 8, 8, 8]; % horizontal position
y = [ 5, 2, 6, 3, 4, 7]; % vertical position

for n = 1: length (x)
  distMat(y(n), x(n)) = 0; % set object pixels
end

% assign distances column-wise
for j = 1:L % for all columns
  distStep = 1;
  for i = 2:M % propagate distances downwards
    if distMat(i,j) > distMat(i-1,j) + distStep
      distMat(i,j) = distMat(i-1,j) + distStep;
      distStep = distStep + 2;
    else
      distStep = 1;
    end
  end
  distStep = 1;
  for i = M-1:-1:1 % propagate distances upwards
    if distMat(i,j) > distMat(i+1,j) + distStep
      distMat(i,j) = distMat(i+1,j) + distStep;
      distStep = distStep + 2;
    else
      distStep = 1;
    end
  end
end

% determine distances row-wise
for i = 1:M % all rows
  % copy row of vertical distances
  distMatV = distMat(i,:);
  for j = 1:L % column positions
    % initialize minimum distance
    distMin = distMatV(j);
    for k = 1:L % compare to column positions
      % combine vert. with horiz. component
      dist = distMatV(k) + (k-j)*(k-j);
      if distMin > dist
        distMin = dist; % new minimum distance
      end
    end
    distMat(i,j) = distMin; % assign minimum
  end
end

distMat % outputs the resulting squared distances
```

Listing 8: Improved row processing of simple EEDT algorithm

```
 initialization
 :
determination of vertical distances
 :
% determine distances row-wise
for i = 1:M % all rows
  % copy row of vertical distances
  distMatV = distMat(i,:);
  for j = 1:L % column positions j0
    % initialize minimum distance
    distMin = distMatV(j);
    for k = j+1:L % compare to column positions forward
      if distMatV(k) < maxDist
        % combine vert. with horiz. component
        distHor = (k-j)*(k-j);
        if distHor >= distMin
          break; % pure horizontal component is longer then current distance
        end
        dist = distMatV(k) + distHor;
        if distMin > dist
          distMin = dist; % store new minimum distance
        end
      end
    end
    for k = j-1:-1:1 % compare to column positions backward
      if distMatV(k) < maxDist
        % combine vertical with horizontal component
        distHor = (k-j)*(k-j);
        if distHor >= distMin
          break; % pure horizontal component ...
        end
        dist = distMatV(k) + distHor;
        if distMin > dist
          distMin = dist; % store new minimum distance
        end
      end
    end
    distMat(i,j) = distMin; % assign minimum
  end
end

distMat % outputs the resulting squared distances
```

Listing 9: Fast algorithm for EEDT

```
initialization
:
determination  of  vertical  distances
:
% determine distances row-wise
js  = zeros( 1,  L+1); % stores intersection positions
ks  = zeros( 1,  L); %  positions of contributors
for  i  = 1:M % all rows
  distMatV = distMat( i ,:) ; %copy row of distances
  idx  = 1;
  js (1)  = −maxDist; % serves as stopping point
  ks(1)  = 1; % assume first (possibly dummy) contributor
  m = 1; % take parabola at first column (if any) as given
  while  m < L % look for next contributor
    m = m + 1;
    if  distMatV(m) < maxDist
      mm = m ∗ m;
      % compute intersection with previous contributor
      k  = ks(idx); % position of previous
      j  = ceil (  (distMatV(m) − distMatV(k) − k∗k + mm) / ( 2∗(m−k) ));
      while  j  <= js(idx) % new parabola hides previous
        idx  = idx − 1; % go one element back
        k  = ks(idx); % position of previous
        % compute intersection with previous contributor
        j  = ceil (  (distMatV(m) − distMatV(k) − k∗k + mm) / ( 2∗(m−k) ));
      end
      if  j  <= L % make sure that new parabola contributes inside matrix
        idx  = idx + 1; % store parameters in next elements
        js ( idx )  = max(1,j); % save new intersection, keep it inside range
        ks( idx )  = m; % save column of next contributor
      end
    end
  end
  js (1)  = 1; % left border of contribution
  js ( idx+1)  = L+1; % right border of contribution
  % now apply collected information (lower envelop)
  for  n  = 1:idx % for all stored contributors
    k  = ks(n); % column of contributor
    for  j  = js(n)  :  js (n+1)−1 % get region of contribution
      distMat( i , j )  = distMatV(k) + (j−k) ∗ (j−k); % assign distance
    end
  end
end

distMat % outputs the resulting squared distances
```

References

[Bai05] Bailey D.G.: An efficient Euclidean distance transform. R. Klette; J. Žunić (Editors), *Combinatorial Image Analysis*, Springer, Berlin, Heidelberg, 2005, 394–408

[Bor84] Borgefors G.: Distance transformations in arbitrary dimensions. *Computer Vision, Graphics, and Image Processing*, vol. 27, 1984, 321 – 345

[Dan80] Danielsson P.E.: Euclidean distance mapping. *Computer Graphics and Image Processing*, vol. 14, no. 3, 1980, 227–248, URL `https://www.sciencedirect.com/science/article/pii/0146664X80900544`

[Fab08] Fabbri R.; Costa L.D.F.; Torelli J.C.; Bruno O.M.: 2D Euclidean distance transform algorithms: A comparative survey. *ACM Comput. Surv.*, vol. 40, no. 1, February 2008, URL `https://doi.org/10.1145/1322432.1322434`

[Fel12] Felzenszwalb P.; Huttenlocher D.: Distance transforms of sampled functions. *Theory of Computing*, vol. 8, September 2012, 415 – 428

[Gre07] Grevera G.J.: Distance transform algorithms and their implementation and evaluation. *Deformable Models: Biomedical and Clinical Applications*, Springer New York, New York, NY, 2007, 33–60, URL `https://doi.org/10.1007/978-0-387-68413-0_2`

[Mau03] Maurer C.; Qi R.; Raghavan V.: A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 2, 2003, 265–270

[Mei00] Meijster A.; Roerdink J.; Hesselink W.: A general algorithm for computing distance transforms in linear time. J. Goutsias; L. Vincent; D. Bloomberg (Editors), *Mathematical morphology and its applications to image and signal processing*, Computational imaging and vision, University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science, 2000, 331–340, relation: http://www.rug.nl/informatica/onderzoek/bernoulli; 5th International Symposium on Mathematical Morphology (ISMM); Conference date: 26-06-2000 Through 29-06-2000

[Ros68] Rosenfeld A.; Pfaltz J.: Distance functions on digital pictures. *Pattern Recognition*, vol. 1, 1968, 33 – 61