Fast Exact Euclidean Distance (FEED): A new class of adaptable distance transforms

Theo E. Schouten and Egon L. van den Broek, *Member, IEEE*

Abstract—A new unique class of foldable distance transforms of digital images (DT) is introduced, baptized: Fast Exact Euclidean Distance (FEED) transforms. FEED class algorithms calculate the DT starting directly from the definition or rather its *inverse*. The principle of FEED class algorithms is introduced, followed by strategies for their efficient implementation. It is shown that FEED class algorithms unite properties of ordered propagation, raster scanning, and independent scanning DT. Moreover, FEED class algorithms shown to have a unique property: they can be tailored to the images under investigation. Benchmarks are conducted on both the Fabbri et al. data set and on a newly developed data set. Three baseline, three approximate, and three state-of-the-art DT algorithms were included, in addition to two implementations of FEED class algorithms. It illustrates that FEED class algorithms i) provide truly exact Euclidean DT; ii) do no suffer from disconnected Voronoi tiles, which is a unique feature for non-parallel but fast DT; iii) outperform any other approximate and exact Euclidean DT with its time complexity $O(N)$, even after their optimization; and iv) are unequaled in that they can be adapted to the characteristics of the image class at hand. The source code of all algorithms included as well as the data sets used for both benchmarks are provided as supplementary material to this article.

Index Terms—Fast Exact Euclidean Distance (FEED), distance transform, distance transformation, Voronoi, computational complexity, adaptive, benchmark

1 Introduction

Until today, throughout almost half a century since the pioneering work of Azriel Rosenfeld and John L. Pfaltz [1], [2], research on distance transformation (DT) has remained challenging and continuously new methods and algorithms are being proposed (cf. [3]). In parallel with research on the fundaments of DT, a still continuing avalanche of applications emerged, which emphasizes the ever remaining importance of DT, also in current image processing. DT can be applied in a range of settings, either on their own or as an important intermediate or auxiliary method in many applications [4], [5], [6], applied by both academics and industry [7]. DT have been applied for (robot) navigation and trajectory planning [4], [8], sensor networks [9], tracking [10], and, in particular, biomedical image analysis [4], [11].

A DT [1] calculates an image, also called a distance map (DM). Distance is a fundamental notion with such functions. Therefore, we first define the L_p distance metric:

$$d_p(\mathbf{x},\mathbf{y}) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}, \qquad (1)$$

where $\mathbf{x}$ and $\mathbf{y}$ are n-tuples, i is used to denote their n coordinates (or dimensions), and $1 \leq p \leq \infty$ [12], [21], [23]. The L_p distance metric can be defined in an n-dimensional space (see (1)). With DT often a two-dimensional (2D) [26] or three-dimensional (3D) space is required [27], [28], as most digital images are 2D or 3D. Moreover, most applications that use DT run on sequential architectures (cf. [29]). Therefore, we limit this article to 2D DT on sequential architectures. The value of each pixel p is its distance (D; i.e., according to a given distance metric, see (1)) to a given set of pixels O in the original binary image:

$$\mathrm{DM}(p) = \min\{\mathrm{D}(p,q), q \in O\}. \qquad (2)$$

The set pixels here is called here O because it often consists of the pixels of objects. However, it might as well consist of either background pixels or data points in a certain feature space. Although (2) is straightforward it is hard to develop an algorithm with a low time complexity [5], [30], [31]. The number of operations depends on the image content in addition to the image size and results to a time complexity up to $O(N^2)$ [5], with $N = n \times m$, n and m being the size of the two dimensions of a 2D image.

DT algorithms can be classified in 3 broad classes, similar to mathematical morphology algorithms. This taxonomy is adopted from Fabbri, da F. Costa, Torelli, and Bruno [5]. Its classification of DT algorithms is determined by the order in which they process pixels:

1) ordered propagation ($\mathcal{P}$): Computation of the minimal distance starting from the object pixels and progressively determining new distance values to background pixels in order of increasing distance.

- *Theo E. Schouten is with the Institute for Computing and Information Science (ICIS), Radboud University and with his consultancy company theos.nl, Dorpstraat 23, 6691 AV Gendt, The Netherlands. E-mails: ths@cs.ru.nl and theo@theos.nl.*
- *Egon L. van den Broek is with the Department of Information and Computing Sciences, Utrecht University; Human Media Interaction (HMI), University of Twente; and Karakter University Center, Radboud University Medical Center Nijmegen, all The Netherlands. Homepage: http://www.human-centeredcomputing.com/. E-mail: vandenbroek@acm.org*

TABLE 1
Ten distance transforms (DT), including $i)$ three baseline DT (β) that are well known baseline algorithms, $ii)$ three approximate Euclidean DT (EDT) (α), and $iii)$ four exact EDT (ϵ), including the FEED class algorithms. CH11 and CH34 denote respectively the city block (or Chamfer 1,1 distance) and the Chamfer 3,4 distance [12], [13]. A brief introduction to them is provided including their reference, the author(s), type (T), class (C) code (name), a brief description, and notes. The DT classes $\mathcal{R}$, $\mathcal{P}$, and $\mathcal{S}$ denote respectively: raster scanning, ordered propagation, and independent scanning (see also Section 1). The source code of all algorithms described here is provided as supplementary material to this article.

reference			T	C	code	description and notes
[1] [2]	1966 1968	Rosenfeld & Pfaltz	β	$\mathcal{R}$	CH11	One of the first DT, which marked the start of a new field of research. It provides a crude but fast approximation of the ED. See also Section 1.
[14]	1980	Danielsson	α	$\mathcal{R}$	4SED	A vector value is used during both initialization and the two scans over the image. The norm of the vector denotes its distance. The two scans both require three passes over each row. Danielsson himself showed that in the worst case his algorithm only results in an error, which is a fraction of the grid constant.
[12], [13]	1984 1986	Borgefors	β	$\mathcal{R}$	CH34	An instance of a well known class of non-exact DT, which has proved to be very fast. It can be optimized on the application at hand and the trade off between computational complexity and accuracy. Consequently, this class is still among the most often used DT [7].
[15]	1988	Ye	α	$\mathcal{R}$	4SED$^+$	Implementation improvement of Danielsson's 4SED [14], which saves the squared integers that are needed for the minimum operator in a separate matrix.
[16]	1998	Coiras, Santamaria, & Miravet	β	$\mathcal{P}$	HexaD	A combination of city-block (CH11) and chessboard growth (cf. [1], [2]). It provided an algorithm for the empirical hexadecagonal growth presented in [17]. It approximates the EDT better than the Chamfer 5,7,11 model [13].
[6] [11]	2003	Maurer, Jr., Qi and Raghavan	ϵ	$\mathcal{S}$	Maurer	See Section 1 for its description. *All* speed optimalizations proposed in [6] have been implemented.
[18] [19]	2004	Shih & Wu	α	$\mathcal{R}$	EDT-2	Uses 2 scans with a 3×3 neighborhood over the image. During the scans it saves squares of intermediate EDs for each pixel. The authors claimed that their algorithm provided exact EDT; however, Van den Broek et al. [7], [20] showed that their can algorithm can provide errors.
[21]	2007	Coeurjolly and Montanvert	ϵ	$\mathcal{S}$	SEDT	See Section 1 for its description. Originally independently developed by Hirata [22] and Meijster et al. [23].
[24]	2009	Lucet	ϵ	$\mathcal{S}$	LLT*	See Section 1 for its description. Optimized implementation; that is, the division by 2 is replaced by multiplication elsewhere; consequently, all the calculations could be done in integer arithmetic.
[25]	*2004*	Schouten and Van den Broek	ϵ	!	FEED	New class of 2D EDT briefly introduced in [25] and properly introduced in the current article.

2) raster scanning ($\mathcal{R}$): 2D masks are used to process pixels line by line, top to bottom and, subsequently, in reversed order.
3) independent scanning ($\mathcal{S}$): Each row of the image is processed independently of each other. Subsequently, each column of that image is processed independently of the other columns, to produce the final DT. Note that the values of the intermediate image cannot be regarded as distances.

For an exhaustive survey on 2D DT, we refer to Fabbri, da F. Costa, Torelli, and Bruno [5] or, alternatively, for a concise definition of DT and their main properties, we refer to Section 2 of Maurer, Qi, and Raghavan [6]. For reasons of brevity, we will refrain from repeating this work. In contrast, this article introduces an alternative approach for Euclidean DT (EDT), as was introduced by Schouten and Van den Broek [25]: the Fast Exact Euclidean Distance (FEED) transformation. As we will explain in this article, this EDT cannot be assigned to one of these three classes of DT and, as such, FEED launches a new class of DT.

To enable a proper positioning of the work presented in this article, we will denote the field's origin and introduce some of its baseline DT, which are still used frequently and, par excellence, are examples of raster scanning ($\mathcal{R}$). Next, we will briefly touch the less frequently investigated ordered propagation algorithms ($\mathcal{P}$). Last, we will discuss three state-of-the-art independent scanning algorithms ($\mathcal{S}$), as were presented in the last decade. For complete structured overviews of the work on DT, we refer to recent surveys such as that of Fabbri, da F. Costa, Torelli, and Bruno [5] and, for an industrial perspective, that of Van den Broek and Schouten [7]. For an overview of the DT employed in the current article, we refer to Table 1.

Rosenfeld and Pfaltz [1] introduced the first fast algorithms for the city-block and chessboard distance metrics in 1966. Two decades later, Borgefors [13] extended them to Chamfer DT, which provide better approximations to the L_2 metric. These three metrics have in common that they all use raster scans over the image to propagate distance using local information only. However, as the tiles of the Voronoi diagram are not always connected sets on a discrete lattice [32], the (exact) Euclidean Distance transform (EDT) cannot be obtained by raster scans. Therefore, more complex propagation methods have been developed to obtain the golden standard: the EDT (e.g., [4], [16]). The Euclidean metric (d_E) is directly derived from (1) with $p = 2$. Finding the DT with respect to the Euclidean metric is, even in 2D, rather time consuming (cf. [30], [31]). However, throughout the last decade several algorithms have been introduced that provided the EDT in reasonable time (e.g., see Table 1 and [5], [7]).

In 2003, Maurer, Qi and Raghavan [6], [11] obtained an exact EDT, based on dimensionality reduction and partial Voronoi diagram construction (cf. [33]). Their work improved on similar work published 8 and 5 years before [34], [35], [36]. In 2007, Coeurjolly and Montanvert [21] presented algorithms to solve the reverse EDT, again similar to the work just mentioned. More recently, in 2009, Lucet [24] presented several sequential exact EDT algorithms, based on fundamental transforms of convex analysis (e.g., the LLT algorithm). These approaches all use dimensional decomposition, that is they start by processing each row independently of the other rows. Hence, these recent algorithms all belong to the class of independent scanning DT.

This article continues the work on FEED [25] in Sections 2–5 and, as such, introduces a new class of DT. First, in Section 2, we will introduce the principle of FEED. Next, in Section 3, several strategies for the implementation of FEED class algorithms will be introduced. Section 4 will introduce the generic FEED class of algorithms followed by Section 5 that will show how FEED class algorithms can be adapted to the set of images at hand. With execution time / complexity and precision being the core features of EDT, both are assessed in an exhaustive benchmark, which is presented in Section 6. This benchmark includes the most important DT claimed to be both fast and exact and compares them with the two FEED class algorithms introduced in the previous section. We will end this article in Section 10 with a concise discussion.

2 Principle of FEED class algorithms

Let us define a binary $\{0,1\}$ matrix $I(p)$ with $p \in \{1,\dots,n\} \times \{1,\dots,m\}$, $O = \{p \mid I(p) = 0\}$ being the set of object pixels, and $B = \{p \mid I(p) = 1\}$ being the set of background pixels. Then, according to (2), the squared ED map (EDM^2) of I is the $n \times m$ matrix

$$\text{EDM}^2(p) = \min_q \{||p - q||^2 \mid q \in O\}. \quad (3)$$

In the FEED class, this is achieved by letting each object pixel feed its ED^2 to all pixels in the image and letting each pixel in the image take the minimum of all the received values. Its naive implementation is as follows:

```
init:       EDM^2(p) = if $p \in O$ then $0$ else $n^2 + m^2$
feed:       for each $q \in O$
receive:        for each $p \in I$
                    $\text{EDM}^2(p) = \min(\text{EDM}^2(p), \|p - q\|^2)$.
                                                                    (4)
```

This naive algorithm achieves the correct result; but, it is also very slow. Fortunately, several methods are available to speed up this naive algorithm considerably. The three basic methods are the following:

- When all the four 4-connected neighbors of a $q \in O$ are also object pixels (i.e., $\in O$); then, each background pixel is closer to one of those neighbors than to q itself. Therefore, one can restrict the feeding pixels in (4) to the border pixels of O, denoted by $B(O)$ (i.e., those object pixels that have at least one of their 4-connected neighbors in the background).
- Further consider a $b \in B(O)$ and let $q \in O$ be any other object pixel, then the receiving pixels p in (4) can be restricted to those with property $||p - b||^2 \leq ||p - q||^2$. Those pixels p are on or on one side of a straight line: the bisection line between b and q, which is illustrated in Fig. 1(a). By taking other object pixels q into account, the pixels p to which b has to feed can be restricted to those that lie within a certain area A_b, see Fig. 1(b) for a graphical explanation of this.
- In principle, each area A_b can be minimized, containing solely the background pixels, which have smaller or equal distance to the considered $b \in B(O)$ than to any other object pixel. However, in general, the time needed to locate these areas will be larger than the time gained by having to feed fewer pixels, because each feed requires only a small number of operations. Therefore, the process of taking more and more qs into account to minimize an area has to be stopped when the area is small enough. That is, when a further reduction does not decrease the execution time. Moreover, the order in which the qs are considered is important for the execution time, the ones closer to the minimal area give larger size reductions of A_b.

With these three speed ups taken into consideration, we define the basic FEED class:

```
init:       EDM^2(p) = if $p \in O$ then $0$ else $n^2 + m^2$
feed:       for each $b \in B(O)$
receive:        determine $a$ suitable $A_b$
update:             for each $p \in A_b$
                        $\text{EDM}^2(p) = \min(\text{EDM}^2(p), \|p - b\|^2)$
                                                                    (5)
```

The determination of a suitable A_b requires the development of strategies to search for the object pixels qs that give the largest reduction of A_b. The gain that can be achieved in this way depends on the distribution of the object pixels over the image. The same is valid for the parameters that determine when to stop reducing the size of A_b. The most efficient strategy has to be experimentally determined on a representative sample of the kind of images one wants to process for a given application. Hence, from the basic FEED class defined in (5), various distinct FEED class algorithms can be derived. These tailored FEED class algorithms, all providing exact EDMs, are adapted to the type of images such that their execution time is minimized.

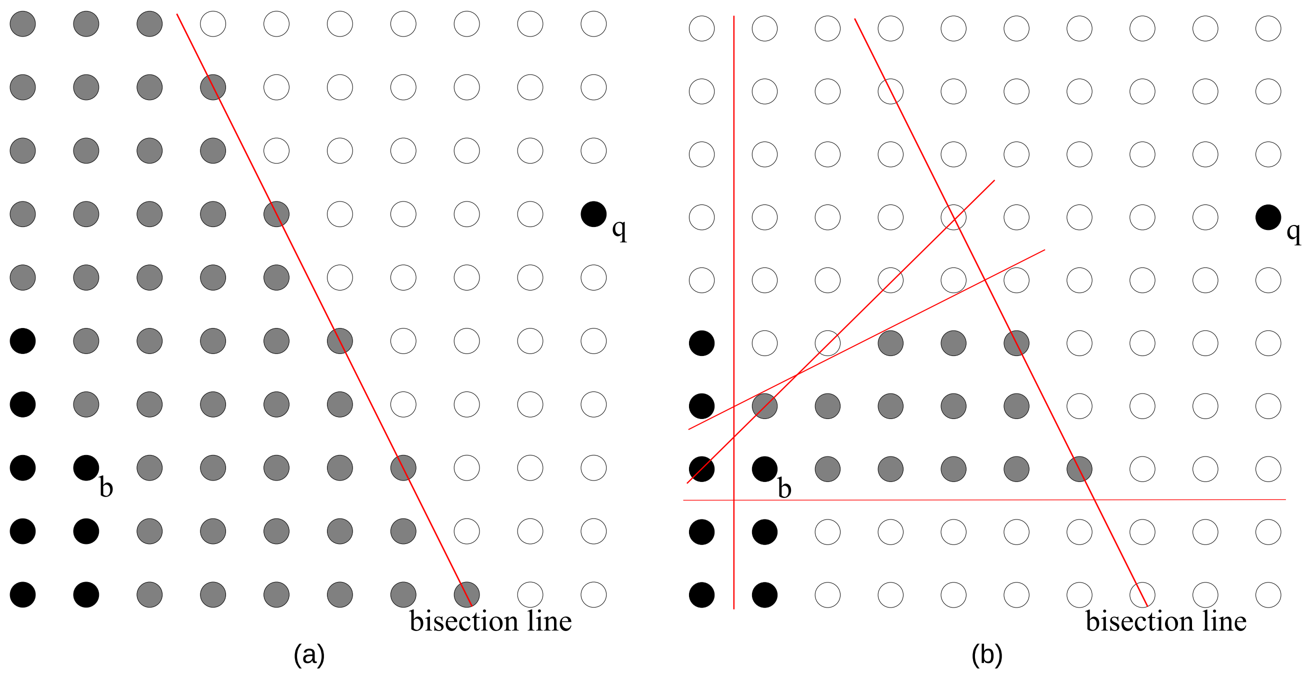


Fig. 1. Using the rotational invariance of the ED^2 metric. Black circles indicate object pixels, the other circles are background pixels. (a) Only pixels on and to the left of the bisection line between the border object pixel b and the object pixel q have to be fed by b, they are indicated by gray circles. (b) Using 4 additional bisection lines.

3 Strategies for FEED class algorithms

In this section, we will describe a number of strategies for FEED class algorithms. Both the merit and the implementation of these strategies will be discussed.

3.1 Bisection lines

For each border pixel b, a local (x, y) coordinate system is defined, with the origin in b, the x axis aligned with the rows and the center of each pixel being on integer coordinates (i, j). An object pixel q then gives the bisection line defined by:

$$2q_ix + 2q_jy = (q_i^2 + q_j^2). \tag{6}$$

So, a bisection line can simply be presented by the pair (q_i, q_j). For the basic FEED class (5), only the integer coordinates of the pixels are of importance. Further, the pixels on the bisection line defined by (6) need to be fed by b xor q, not by both of them. For example, a test on the y coordinate of q can achieve this. This leads to an adapted equation for the bisection line:

$$\begin{array}{rcl} q_ix + q_jy & = & [int]\ (q_i^2 + q_j^2 + ud)\ /\ 2 \\ ud & = & \text{if } q_j \geq 0 \text{ then } 0 \text{ else } -1, \end{array} \tag{7}$$

where $[int]$ is the truncation to integer values and ud abbreviates *up-down*.

Pixels p that satisfy $D(p, b) \leq D(p, q)$ need to be fed by b. Subsequently, they are defined by:

$$q_ip_i + q_jp_j \leq [int]\ (q_i^2 + q_j^2 + ud)\ /\ 2. \tag{8}$$

Note that these equations also imply that *all* calculations for FEED class algorithms can be done in integer.

3.2 Line search strategy

The search for object pixels q is done along search lines starting at b defined by $(i, j) = k\ (m_i, m_j)$, with m_i and m_j being minimal integers and k the running (integer) index starting at 1. Then, the equation of the bisection line for an object pixel q at $k(m_i, m_j)$ becomes:

$$m_ix + m_jy = [int](\ k\ (m_i^2 + m_j^2) + ud)\ /\ 2. \tag{9}$$

As soon as an object pixel is found at a certain k, further searching along the line can be stopped. Additional object points would result in bisection lines that are parallel to the first bisection line; hence, they can not decrease A_b further. As an example, in Fig. 1(b) the object pixel q is on the search line $(i, j) = k(2, 1)$ at $k = 4$. Note that not all the pixels on a search line have to be checked for being an object pixel. An informed search can be employed using certain step size, depending on the expected size of the objects in the images.

Search lines are usually employed in groups, derived from a given positive n consisting of the lines $k(\pm n, \pm m)$ and $k(\pm m, \pm n)$ with $0 < m < n$ and m and n have no common divisor other than 1. Such a group is further denoted by $k\{m\}$; for example, $k\{4\}$ consists of the lines $k(\pm 4, \pm 1)$,$k(\pm 1, \pm 4)$, $k(\pm 4, \pm 3)$ and $k(\pm 3, \pm 4)$.

Searching along some chosen search lines is usually divided into two steps: $i)$ In a small area around b because object pixels found there give the largest reduction in A_b and $ii)$ When the size of A_b is not small enough, possibly after other searches, the search is further continued along the chosen search lines.

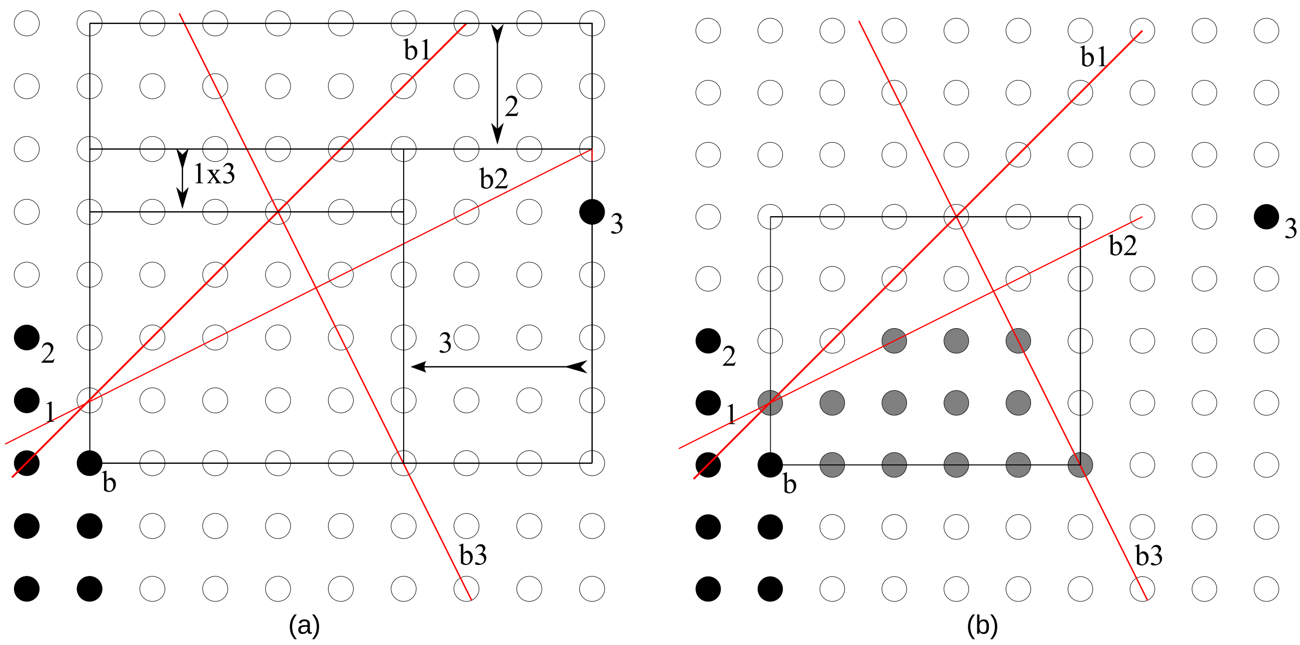


Fig. 2. Using the bounding box bb. (a) The large rectangle is the original bb. Bisection line $b1$ between object pixels b and 1 does not change bb. Bisection line $b2$ decreases the size of bb as indicated by the arrow 2. Also $b3$ changes bb. The intersection between $b1$ and $b3$ also decreases the size of bb. (b) The filling process: for each scanline in bb each bisection line defines a range of pixels where b should feed to. The gray pixels are fed by b.

3.3 Bounding box

For various reasons, it is efficient to keep a bounding box bb around (the possibly complex shaped) A_b, instead of using an exact representation. The area of a bb can be calculated quickly and can be used to determine whether or not to stop the process of finding a smaller A_b. Searching along a line can be stopped when it is not possible to further reduce the size of A_b (see Section 3.2). The fastest check for this is to use the corner of bb in the same quadrant around b as where the search line is.

For each new bisection line, its intersections with bb can be quickly calculated to determine whether or not it reduces the size of bb. This is graphically illustrated in Fig. 2(a), where bisection lines $b2$ and $b3$ change bb but $b1$ does not. Note that after the change of bb by $b3$, $b2$ could have been used to reduce bb further. Also intersections between bisection lines can be used to reduce bb as is shown in Fig. 2(a) by the intersection of $b1$ and $b3$. The bb is also used in the receiving and update steps of (5), as is illustrated in Fig. 2(b). For each row, the intersections with the bisection lines are used to determine the pixels to be fed by b. Thus, only the pixels inside the complex shaped A_b are filled. Moreover, not all the bisections lines have to be used, which enables the minimization of FEED class algorithms' total execution time. When bb is small enough, it is faster not to use the bisection lines; hence, the whole rectangular shaped bb is filled.

3.4 Border pixels

Border pixels are located during a single raster scan over the image. A border pixel b is only processed after a possible next one in the same row is determined. Then, there is sufficient information available for a first determination of the horizontal extent of the first bb.

An auxiliary vector can be used to determine the vertical extent of bb. Then, for each column of this vector, the row number of the last processed border pixel is remembered. Together with the status of the pixel directly below the border pixel this gives a fast determination of the lower side of bb. For locating the top of bb, a line search in the vertical direction ($k(0, 1)$), which to be made that can be divided into two steps.

3.5 Quadrant search strategy

For images with a low percentage of object pixels (e.g., random dot images) or object images with a low number of small objects, the line search strategy alone might not be very efficient. For such images, the quadrant search strategy can be used effectively.

A search is performed along the horizontal line through b using increasing distances x from b, for object pixels in the vertical direction. Auxiliary vectors are used to store object pixels both below and above the b's row (see Section 3.4). For each quadrant around b, the object pixel closest to b is kept. The search in a quadrant is stopped when a new x cannot either improve on the current closest object pixel or reduce the bounding box bb. Subsequently, the intersections of the (maximal 4) bisection lines found are used to decrease the size of bb. Last, the receiving and update steps of (5) are performed.

Note that this set of strategies is not exhaustive. Many other (additional) strategies can be explored. For example, i) object representation during the initialization phase can be utilized to minimize bb; ii) for particular shapes of A_b (e.g., single lines in horizontal or vertical direction), faster filling methods can be used; and iii) the correlations between border pixels can be exploited.

4 The FEED class

Algorithm 1 General FEED class algorithm.

```
 1: Initialize EDM$^2$
 2: for each border found in the raster scan do
 3:   determine first $bb$
 4:   update auxiliary vectors
 5:   if $bb$ is a vertical or horizontal line then
 6:     FILL single line && continue loop.
 7:   end if
 8:   if no top object pixel found for first $bb$ then
 9:     continue search for top of $bb$
10:     update second auxiliary vector
11:   end if
12:   if $size(bb) < P_1$ then
13:     FILL $bb$ && continue loop.
14:   end if
15:   Search $k\{1\}$, with $k \leq 4$
16:   if any object pixel found then
17:     if $size(bb) < P_2$ then
18:       FILL $bb$ && continue loop.
19:     end if
20:     if $A_b$ is diagonal and $size(bb) < P_3$ then
21:       FILL diagonal && continue loop.
22:     end if
23:     Search $k\{2\}$ with $k = 1$
24:     if $size(bb) < P_4$ then
25:       FILL $bb$ && continue loop.
26:     end if
27:   else
28:     Search $k\{2\}$ with $k = 1$
29:     if $size(bb) < P_5$ then
30:       FILL $bb$ && continue loop.
31:     end if
32:   end if
33:   Perform quadrant search
34:   if $size(bb) < P_6$ then
35:     FILL $bb$
36:   else
37:     FILL $A_b$
38:   end if
39: end for
```

In Algorithm 1, the general FEED class is presented. Its source code is provided as supplementary material to this article. Section 3.4 already described both the localization of the border pixels (line 2) and the determination of the first bb (line 3). The latter uses the variant of checking only the first 4 pixels above the border pixel. Subsequently, a possible single line bb is handled in lines $5-7$, as was mentioned at the end of Section 3.

In lines $8-10$ the search for the top of the bb is continued if needed. The "Search $k\{n\}$" operations are line searches, as described in Section 3.2, followed by intersections with the bb in order to (possibly) reduce its size, as described in Section 3.3. Each new bisection line intersects with all existing bisection lines, to check for further reductions of the bb. The split in two paths through the algorithm in line 16 ensures a fast check on the existence of previously found bisection lines.

The "quadrant search" operation in line 33 and its auxiliary vectors (lines 4 and 10) are described in Section 3.5. Note that the first auxiliary vector is also used for a fast determination of the bottom of the first bb as described in Section 3.4 The FILL operations of a bb, a A_b, a single line, or a diagonal shaped A_b are described in Section 3.3.

The $P_i; i = 1\ to\ 6$ in Algorithm 1 are parameters, which were adjusted to minimize the total execution time. The configuration of P_i of FEED algorithms is best done by exploiting the geometrical properties of the image content at hand to reduce the 6D search space. In practice, this will often require a manual procedure of setting all six parameters sequentially and, subsequently, do the same in the reversed manner. Alternatively, if needed (or preferred), a brute force search strategy can be used on a representative sample of the image set to determine the parameter settings. Obviously, these strategies can also be combined: The brute force strategy can be used on top of the manual specification of the parameters, using a window surrounding them.

5 FEED: A class of adaptive DT

The characteristics of object like images can be employed to developed a specialized and faster FEED class algorithm for them. The objects have a certain extent and, thus, the line search strategy (see Section 3.2), can be effectively used more than for the general usable FEED class algorithms. Also the rather time consuming quadrant search strategy (see Section 3.5) can be avoided.

5.1 The general principle

Object like images often contain a low number of border pixels. This means that the initialization step in (5) can be combined with determining the border pixels together with their first bbs, as described in Section 3.4. This information is stored in a list of borders, containing for each border its position and the initial bb size, which requires only a minimum amount of additional memory. Consequently, no scan in the vertical direction has to be made to locate the top of a bb (cf. Section 3.4).

An auxiliary vector can be used to store for each column a reference to the last border in that column in the border list. Subsequently, the top of the border's bb is filled in when the next border in the column is found or when there is none. Further, it appeared to be time effective to initialize each background pixel with the ED^2 to the closest border pixel in its row, instead of with the maximal possible ED^2 in the image. In the case of a row

Algorithm 2 FEED for object like images.

```
Initialization combined with finding borders
for all borders in the border list do
    if $bb$ is a vertical line then
        FILL single line && continue loop.
    end if
    if $size(bb) < P_1$ then
        FILL $bb$ && continue loop.
    end if
    Search $k\{1\}$, with $k \leq 4$
    if $A_b$ is diagonal and $size(bb) < P_2$ then
        FILL diagonal && continue loop.
    end if
    if $size(bb) < P_3$ then
        FILL $bb$ && continue loop.
    end if
    Search $k\{2\}$ with $k \leq 2$
    Continue search $k\{1\}$, with $k = 5$ $step$ $P_4$
    if $A_b$ is diagonal and $size(bb) < P_5$ then
        FILL diagonal && continue loop.
    end if
    if $size(bb) < P_5$ then
        FILL $A_b$ && continue loop.
    end if
    Search $k\{4\}$ with $k = 1$
    Continue search $k\{2\}$ with $k = 3$ $step$ $P_6$
    if $A_b$ is diagonal and $size(bb) < P_7$ then
        FILL diagonal && continue loop.
    end if
    if $size(bb) < P_7$ then
        FILL $A_b$ && continue loop.
    end if
    Continue search $k\{4\}$ with $k = 2$ $step$ $P_8$
    if $A_b$ is diagonal then
        FILL diagonal
    else
        FILL $A_b$
    end if
end for
```

with no border pixels, row pixels are initialized with the maximum ED^2. This can still be achieved with a single raster scan of the input image and the EDM^2 map; hence, with a limited amount of extra time.

5.2 The algorithm

In Algorithm 2 the FEED class algorithm for object like images is presented. Its source code is provided as supplementary material to this article. The FILL operations of a bb, an A_b, a single line, and a diagonal shaped A_b are described in Section 3.3. The search operations are line searches, as described in Section 3.2, followed by intersections with the bb in order to (possibly) reduce its size, as described in Section 3.3. To check for further reductions of the bb, each new bisection line of the form $k(m_i, m_j)$ intersects with all existing $k(\pm 1, \pm 1)$ bisection lines and with all bisection lines $k(\pm m_i, \pm m_j)$, as described in Section 3.3.

The $P_i; i = 1$ to 8 in Algorithm 2 are parameters that can be adjusted to tailor the algorithm to the image class at hand and, as such, are an important part of the optimization strategy. The values of P_i can be assessed using either a knowledge driven approach or a data driven optimization heuristic. This mainly depends on the images under investigation. See also Section 4 for a brief explanation of this.

6 Benchmarking: Generic optimization strategies

The importance of EDT for pattern analysis and machine intelligence is well illustrated by the impressive number of articles that have appeared on this topic in this journal throughout the last two decades (e.g., [6], [21], [29], [33], [34], [36]). The algorithms each coin some important and unique characteristics (e.g., recently nD [6], [11], time optimization [6], [11], [21], and solving the medial axis extraction [21]). However, work on EDT is seldom compared with alternatives (cf. [6], [7], [11], [21]); the work of Fabbri et al. [5] is an exception on this.

The current article presents a new class of EDT that is not only both fast and exact but can also be tailored to image characteristics. We have set up two benchmarks to compare the proposed FEED class with several alternatives, which will be introduced in Sections 7–9. All algorithms included in these benchmarks have been implemented according to their description, including all known optimization strategies.

The three state-of-the-art algorithms (i.e., Maurer [6], [11], SEDT [21], and LLT* [24]; see Table 1) all use dimensional reduction by first processing one dimension of the input image and, subsequently, the other dimension of the resulting intermediate image. For these algorithms, the fastest implementation is realized through processing first the rows and then the columns. Further, note that the order of processing (i.e., first the rows and then the columns or vice-versa) does not matter for the correct functionality of the three state-of-the-art algorithms; however, it can be of influence on the execution time. To assure the optimal order of processing for all algorithms, the fastest implementation was chosen, which appeared to be the row-column order.

The integration of the initialization step with further processing, resulted in an additional optimization. The background pixels were initialized to a certain value during the processing of the rows. This resulted in a speed up, compared to the originally proposed implementations. As such, this speed up was equivalent to how this is done with the FEED algorithm, as described in Section 3.4. The average speed up achieved was $\pm 14\%$.

All optimizations have been generalized over all algorithms, where possible. Consequently, variance in performance due to differences in optimization strategies was prevented. To aid full control of the claims made here

TABLE 2
The algorithms' average processing time (ave) with their root mean square (rms) in *ns* per pixel on the Fabbri et al. [5] data set of 5 exact DT. See Table 1 for a description of the DT algorithms.

images	specification	Timing (in *ns/pixel*) per algorithm									
		Maurer		LLT*		SEDT		FEED		FEED$_o$	
		ave	rms	ave	rms	ave	rms	ave	rms	ave	rms
turning line	1000×1000 ($0° - 180°$ by $5°$)	33.76	6.13	33.28	6.88	35.45	2.22	42.40	17.80	27.25	11.47
turning line, inverse	1000×1000 ($0° - 180°$ by $5°$)	30.35	0.11	41.66	0.16	32.32	0.08	4.24	0.11	3.76	0.10
turning line	2000×2000 ($0° - 180°$ by $5°$)	35.98	6.36	35.46	6.56	37.67	2.72	73.99	33.83	44.66	20.31
turning line, inverse	2000×2000 ($0° - 180°$ by $5°$)	32.34	0.08	44.01	0.05	35.07	0.07	4.95	0.08	3.78	0.07
inscribed circle	(range: $500 - 4000$)	40.63	0.95	43.58	2.04	41.09	1.99	95.52	45.96	51.43	20.37
inscribed circle, inverse	(range: $500 - 4000$)	34.26	1.48	43.63	2.59	35.68	2.27	9.49	1.30	7.43	0.41
random squares	3000×3000	37.05	2.33	43.95	1.44	38.50	1.33	15.95	10.26	9.99	5.22
random squares,inverse	3000×3000	37.63	2.41	43.84	1.55	38.91	1.39	15.84	8.13	10.44	4.65
random points	1000×1000 ($1\% - 99\%$)	43.25	6.26	49.81	3.89	43.34	4.84	30.41	12.52	34.40	22.46
point in corner	1000×1000 (4 images)	21.62	0.01	19.60	0.00	33.19	2.48	4.79	0.02	5.92	0.01
point in corner, inverse	1000×1000 (4 images)	31.04	0.06	41.60	0.04	31.19	0.03	4.13	0.00	3.75	0.11
half-filled image	1000×1000 (4 images)	28.99	2.34	34.87	4.24	32.68	0.81	5.67	1.21	6.10	2.29
Lenna (or Lena)'s edges	512×512	39.28		37.74		38.37		22.89		20.60	
Lenna (or Lena)'s edges, inverse	512×512	28.52		35.74		29.40		9.54		8.62	

Note. rms is not reported for the Lenna (or Lena)'s edges image as this concerns only one image instead of a set of images.

and facilitate the replication of the results reported here, all source code and test sets are made available as online supplementary material accompanying this article.

7 Benchmarking: Fabbri et al. data set

The data sets described by Fabbri et al. [5] have been adopted and used to compare the exact EDT, as described in Table 1. These concern state-of-the-art algorithms, which provide true exact ED and are the fastest currently available [6], [11], [21], [24]. As such, this replicates the work reported by Fabbri et al. [5], with FEED added to the set of exact EDT. These data sets used are partly available on sourceforge.net[1]. The remaining part of the data set can be easily generated from the description. The results of the benchmark are presented in Table 2. We will now briefly discuss these results.

Fabbri et al. [5]'s turning line data set is par excellence suitable to show the dependence of the execution time on the angle of the border of an object. We generated line images with an angle varying from $0°$ to $180°$ with a step of $5°$. Both FEED implementations show a larger variation in execution time over the angles than the other exact EDT. FEED processes lines under angles of $0°$, $45°$ and $90°$ faster than other angles because each pixel has two opposite 8-connected object pixels. Thus, A_b (i.e.,the area to be filled; see also Section 2) is reduced to a single horizontal or vertical line or two lines under $45°$. For the other angles, A_b is often a much larger, wedge shaped area. Further, the searches in FEED for object pixels at a distance in these kinds of images will be in vain and will only increase the execution time. Hence, FEED will be rather slow on these images, especially when the image size is large. This can be seen in Table 2; cf. the execution times of the turning line images of size 1000×1000 and size 2000×2000.

1. The Fabbri et al. [5] data sets are partly available at http://distance.sourceforge.net/ and partly as supplementary material of this article.

Table 2 also provides the execution times for the inverse images. Then, the turning line is the background for which the EDM have to be determined. With these inverse line images, both FEED versions are incredibly fast compared to the other algorithms. This can be attributed to the fact that FEED is very fast in determining what object pixels should receive an ED of 0.

The inscribed circle images take the circle as background. FEED is not very fast and the execution times are increasing with the increase of image size. This can be explained again by both the object's border that contains all possible angles and the long distances searches that are highly inefficient. In contrast, FEED is again very fast on the inverse images.

Fabbri et al. [5]'s random squares data set combines the variation in angle of the object borders with the variation in percentage of object pixels. Here, FEED also shows a larger variation with both angle and percentage of filling than the other algorithms. However, with this subset, FEED's searches are effective and, consequently, FEED is much faster than the other algorithms. Note that FEED$_o$, the version optimized for object like images, is approximately 30% faster than the general FEED version. The performance on the inverse images is about the same, as is shown in Table 2.

Both FEED and FEED$_o$ show also to be faster than the other algorithms on the random points, point in the corner, and half-filled image (see Table 2). However, with these image sets, FEED showed to be faster than FEED$_o$. To aid the understanding of this effect, please recall (5), which describes that FEED is based on feeding the ED from each boundary pixel b to an area A_b around b. In principle, A_b can be reduced to the minimum; but, that takes more time than gained by having smaller A_bs. Thus, FEED algorithms must use suitable strategies to make the search time for A_bs much smaller than the time gained by having to *feed* smaller A_bs. FEED$_o$ is optimized for images with objects in it. FEED$_o$ makes extensive use of the line search strategy (see Section 3.2) and because

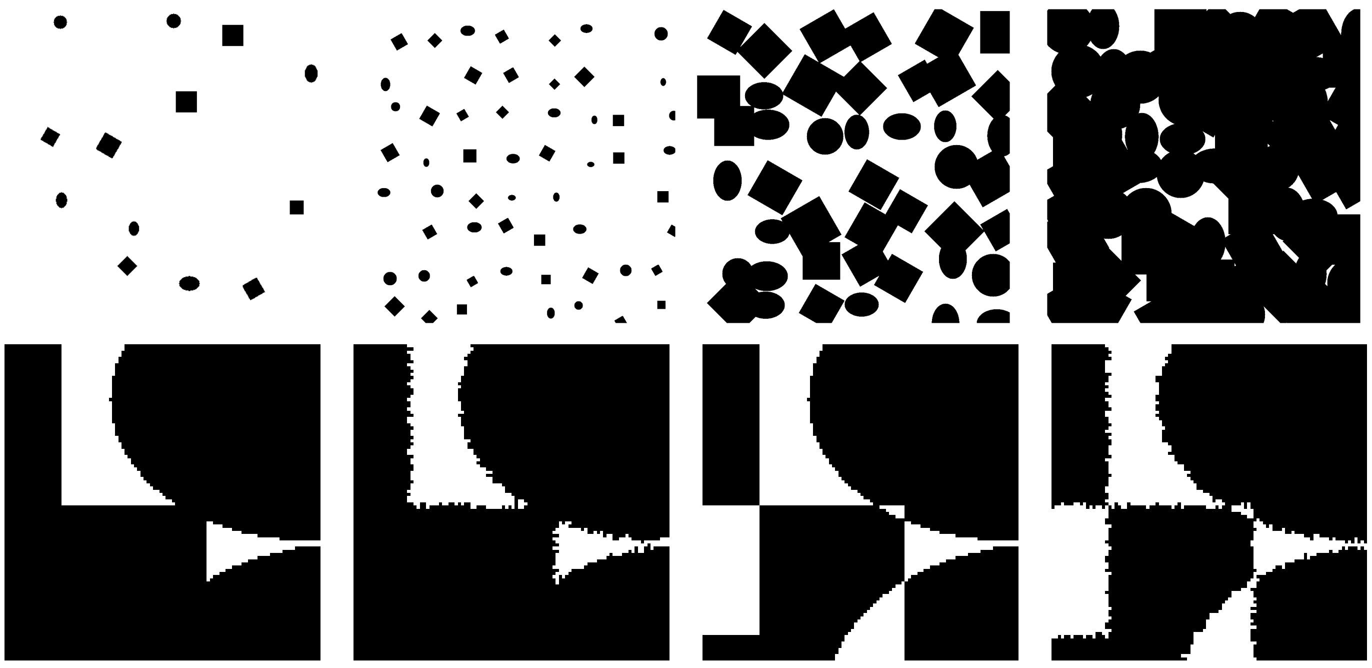


Fig. 3. **Top row:** Four object like images from set O of size 1036×1036, with respectively 1.82%, 5.92%, 50.87%, and 94.08% object pixels. These serve as the foundation of the newly generated data set. **Bottom row:** A sample segment from the third image seen from left of the top row and its three derivatives versions.

objects have a certain size, there is a high chance that a search line hits on an object and, hence, reduce the A_b.

For random point images, the search lines have less chance to hit a random dot. Consequently, Table 2 shows that FEED$_o$ is on average slower than FEED on the used range of random point images (with fillings from 1–99%). Note than the rms for FEED$_o$ is much larger than the rms for FEED. For low fillings FEED$_o$ is much slower than FEED. Above a filling of $\pm 20\%$, FEED$_o$ is faster than FEED. Then, the chance that the search lines hit a random dot is high enough. For both the point in the corner and the half filled images, FEED$_o$ takes longer than FEED. FEED$_o$'s search lines do not hit any object pixel. Moreover, also the quadrant search (see Section 3.5) in FEED does not bring any reduction in A_bs. Nevertheless, both instances of the FEED class are much faster on these images than the other algorithms.

The image with Lenna (or Lena)'s edges is the only realistic image and, as such, perhaps the most interesting one. Table 2 shows that both FEED implementations are fast on the Lenna image and even faster on the inverse of the Lenna image. This suggests that FEED is possibly faster than the other algorithms with (such) realistic object like images, which contain many objects and have a large variation in border angles. To research this claim, the next subsection will present a second benchmark.

Please note that neither FEED nor FEED$_o$ have been tuned for optimal performance on the Fabbri et al. [5] data set. Tailoring a FEED instance to the specific sets of the Fabbri et al. collection would significantly increase FEED's performance but would significantly decline FEED's performance on realistic images, which we consider to be true goal. To assess the algorithms' performance on realistic images, we have developed a data set with balanced properties, which reflect the properties of realistic images. This data set is used with a second, new benchmark, which is presented next.

For now, we conclude that FEED's speed increases both when image's edges are close to $0°$, $45°$, or $90°$ and when the image's object pixels increases. Further, the distribution of the objects over the image area is of influence. In general, long distance searches without the detection of object pixels slows down FEED.

8 Benchmarking: Generation of a new data set

In the previous section, it was shown that the speed of the algorithms depends on the characteristics of the input images, especially the angle of the borders of objects and the percentage of object pixels. This section further and more exhaustively explores characteristics of the EDT under investigation. Therefore, a new test set is developed, with a wide distribution of the object's border angles and the whole range of percentage of object pixels covered. This extensive data set provides realistic object like images, covering the range of characteristics that influences the processing speed. This data set is provided as supplementary material to this article.

To develop the FEED versions, we have generated a set of 32 object like images of size 1036×1036, as follows. A placement grid of 9×9 cells was placed over an empty image. A random number from 15 to 81 objects were

TABLE 3
The complete timing and error results (compared to the Euclidean distance, ED) in seconds of the 11 DT that were included in the benchmark. See Table 1 for a description of the DT algorithms.

Algorithm	Timing (in *ns/pixel*)										Errors compared to (exact) ED for all O				
	O		O'		O_-		O'_-		all O		$\neg$ED	abs. (in pixels)		relative (in %)	
	total	rms	total	rms	total	rms	total	rms	total	rms		ave.	max.	ave.	max.
CH11	5.52	1.22	5.84	1.30	5.93	0.68	6.31	0.64	5.90	0.96	55.36	6.20	135.20	13.24	38.59
4SED	29.70	1.13	29.67	1.11	30.21	0.44	30.19	0.49	29.94	0.81	0.17	δ_a	0.32	δ_r	11.00
CH34	8.79	2.69	9.88	2.97	9.58	1.60	10.85	1.56	9.77	2.21	54.74	0.85	20.41	1.77	5.33
4SED$^+$	21.23	3.69	21.72	3.80	22.58	1.80	23.23	1.65	22.19	2.73	0.17	δ_a	0.32	δ_r	11.00
HexaD	40.42	14.40	41.74	14.98	44.11	9.02	45.57	9.24	42.91	11.91	54.19	0.25	10.06	1.03	38.59
Maurer	32.92	2.33	34.79	2.82	33.86	2.07	34.95	2.34	34.38	2.40	–	–	–	–	–
EDT-2	29.09	9.31	29.86	9.56	31.30	6.23	32.03	6.13	30.57	7.81	5.46	0.06	12.19	0.12	7.50
SEDT	33.17	1.51	35.02	2.15	33.87	0.72	36.01	1.15	34.52	1.38	–	–	–	–	–
LLT*	39.21	4.14	40.37	4.11	39.50	4.38	40.66	4.53	39.93	4.29	–	–	–	–	–
FEED	11.40	2.58	12.42	2.84	12.72	1.19	13.89	1.22	12.61	1.96	–	–	–	–	–
FEED$_o$	7.74	1.85	8.64	2.05	8.72	0.72	9.86	0.97	8.74	1.35	–	–	–	–	–

Abbreviations. total: Total processing time; rms: root mean square; abs.: absolute; ave.: average; and max.: maximum.
Notes. $\neg$ED is reported in % pixels. The exact average errors produced by 4SED [14] and 4SED$^+$ [15] are $\delta_a = 0.000113$ (absolute) and $\delta_r = 0.002459$ (relative). With – is denoted that no (or 0) errors have been generated.

TABLE 4
A one-on-one comparison between all DT algorithms on their average processing time (cf. Table 3).

	CH11	4SED	CH34	4SED$^+$	HexaD	Maurer	EDT-2	SEDT	LLT*	FEED	FEED$_o$
CH11		**5.07**	**1.66**	**3.76**	**7.27**	**5.83**	**5.18**	**5.85**	**6.77**	**2.14**	**1.48**
4SED	0.20		0.33	0.74	1.43	1.15	1.02	1.15	1.33	0.42	0.29
CH34	0.60	3.06		2.27	4.39	3.52	3.13	3.53	4.09	1.29	0.89
4SED$^+$	0.27	1.35	0.44		1.93	1.55	1.38	1.56	1.80	0.57	0.39
HexaD	0.14	0.70	0.23	0.52		0.80	0.71	0.80	0.93	0.29	0.20
Maurer	0.17	0.87	0.28	0.65	1.25		0.89	**1.00**	1.16	0.37	0.25
EDT-2	0.19	0.98	0.32	0.73	1.40	1.12		1.13	1.31	0.41	0.29
SEDT	0.17	0.87	0.28	0.64	1.24	**1.00**	0.89		1.16	0.37	0.25
LLT*	0.15	0.75	0.24	0.56	1.07	0.86	0.77	0.86		0.32	0.22
FEED	**0.47**	**2.37**	**0.77**	**1.76**	**3.40**	**2.73**	**2.42**	**2.74**	**3.17**		**0.69**
FEED$_o$	**0.68**	**3.43**	**1.12**	**2.54**	**4.91**	**3.93**	**3.50**	**3.95**	**4.57**	**1.44**	

randomly placed on the centers of the cells. The centers of the objects were randomly displaced in their cells up to half the width (and height) of a cell. The objects were randomly chosen from a set of 7 objects: a square and its rotations by $30°$, $45°$, and $60°$, a circle, and two ellipses with elongations of a factor 2 in the x respectively y directions. Finally, the sizes of the objects were chosen in ranges such that the percentage of object pixels varied between about 2% and 94%. Four examples of this set are shown in Fig. 3.

For generating the benchmark's data set, a set of 160 object like images O was generated, as described above. From this data set, 3 additional data sets have been derived using either a roughening (O'), an overlap-removal operator (O_-), or both (O'_-). These additional data sets were generated to assess the performance of the algorithms in real world practice, with respect to two dimensions: i) often object borders are not smooth; hence, a roughening operation was applied and ii) objects can show overlap and occlude each other [4] but objects can also have holes or enclosed parts of another matter (e.g., gray and white brain matter [11]). Therefore, an operator to remove object overlaps was applied. Note that the roughening operation generates more object border pixels, with fewer neighbors on average. Examples of these four sets of objects are presented in Fig. 3.

To evaluate the scaling behavior with regard to the image size and, hence, the computational complexity of the algorithms, larger images were generated ranging from 1 to 7 times as large as 1036×1036 (cf. [11]). This was done for the 4 sets of images, which makes a total of 4480 test images. We will denote the combination of these 4 sets as O_s.

9 Benchmarking: Results on the new data set

The complete set of 11 algorithms was included, as are described in Table 1. As such, we aim to provide a concise, representative, and challenging benchmark for FEED class algorithms. The benchmark was conducted on a standard office PC (i.e., Intel® Core 2 Duo E6550 2.33 GHz, 2×32 KBytes L1 and 4096 KBytes L2 cache, and 2 GBytes main memory). The results of the benchmark are presented in Tables 3 and 4. Table 3 provides both the average and the root mean square value (*rms*) execution times in *ns/pixel* for all algorithms on all four data sets as well as on their average. Additionally, Table 3 provides the errors (both in absolute and relative sense) of the baseline and approximate Euclidean DT. Table 4 provides a one-on-one comparison in the processing speed of all algorithms included in the benchmark.

First, we will discuss the results of the different algorithms included in the benchmark and relate them to each other. Second, we will discuss the behavior of the 11 algorithms in relation to the percentage of object pixels present. Third and last, we will discuss the scaling behavior of the algorithms and, consequently, present their computational complexity.

Both FEED and FEED$_o$ require very little processing time, as is illustrated in Fig. 4. When generic FEED is compared with CH11 [1], [2] and CH34 [13], it requires only $2.14\times$ and $1.49\times$ the processing time of CH11 [1], [2] and CH34 [13] respectively. CH11 is the fastest DT algorithm available; however, it is a crude approximation of the EDT, see also Table 3. The FEED algorithm optimized for object images, performs even better and is $1.12\times$ faster than CH34 [13] and only $1.48\times$ slower than CH11 [1], [2]. Both FEED and FEED$_o$ are faster than all other approximate algorithms, see both Tables 3 and 4 and Fig. 4. Taken together, when speed is crucial, CH11 could be a better choice than a FEED class algorithm; however, when precision is also of at least some concern a FEED class choice is by far the best option.

Both FEED and FEED$_o$ are much faster than the other exact EDT, see both Tables 3 and 4 and Fig. 4. FEED is approximately $3\times$ faster than the other three exact EDT (i.e., Maurer [6], [11], SEDT [21], and LLT* [24]). FEED$_o$ is even $1.44\times$ faster than FEED and, as such, respectively $3.93\times, 3.95\times,$ and $4.57\times$ faster than respectively Maurer [6], [11], SEDT [21], and LLT* [24]. Note that Maurer [6], [11] and SEDT [21] do not differ in performance, see also Table 4. Taken together, when an (approximate) exact EDT is needed, a FEED class algorithm, even when not optimized, outperforms all other state-of-the-art algorithms by far.

Where Tables 3 and 4 present average numbers, Fig. 4 presents the performance of the algorithms in relation to the percentage of object pixels in the images. This reveals interesting behavior of the algorithms in relation to the number of object pixels present. HexaD [16] shows a sharp decline in processing time with the increase of the percentage of object pixels, as is shown in Fig. 4. Ye's adaptation [15] of Danielsson's algorithm [14] becomes more fruitful with an increase of object pixels as well. CH11 [1], [2] and CH34 [13] also show a decline in processing time with an increase of object pixels, although for these algorithms the influence of the number of object pixels is limited, see also Fig. 4.

The exact EDT show a different behavior in relation to the percentage of object pixels than the approximate EDT and baseline DT, see Fig. 4. All exact EDT algorithms, except for FEED$_o$ (cf. 4SED$^+$ [15]), show an increase in processing time with an increasing percentage of object pixels, up to approximately 20%. Also with more than 20% object pixels, LLT*'s [24] processing time keeps increasing, where the processing time of Maurer [6], [11], SEDT [21], and FEED starts to decline from that point on, see Fig. 4. The behavior of Maurer [6], [11], SEDT [21], and FEED is similar, although FEED is much faster. FEED$_o$ gradually consumes slightly less processing time when the percentage of object pixels increases.

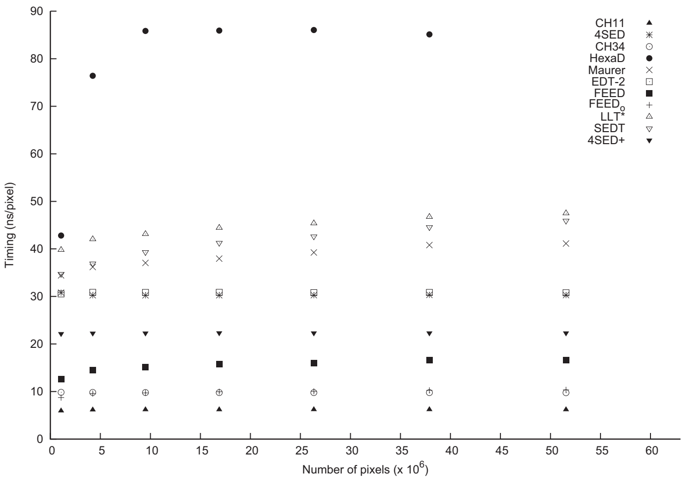
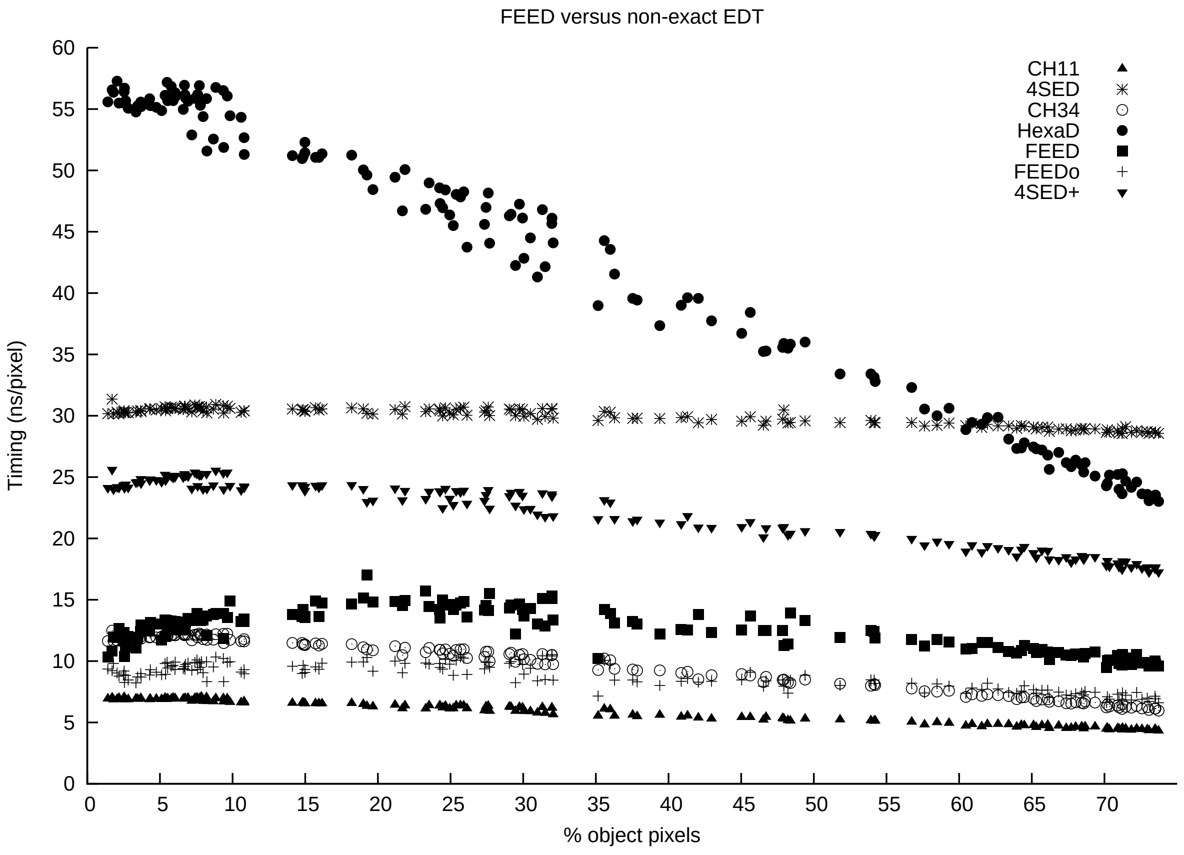


Fig. 5. Execution time as function of the number of pixels (N) in the super set of O_s images.

Fig. 5 gives the scaling behavior of the algorithms, which is the average execution time (in ns) per pixel as function of the size of the images of set O_s. As such, Fig. 5 provides the numerical validation of the time complexity of the algorithms. The processing time of the two baseline DT (i.e., CH11 and CH34) and the approximate EDT 4SED and 4SED+ is a function of the number of pixels (N) in the super set of O_s images, which marks their theoretically optimal $O(N)$ complexity. The HexaD algorithm [16] has a less straight forward time complexity. Initially a sharp increase in processing time is present up to an image size of 10^7 pixels and, subsequently, a gradual decline in processing time is shown. The three approximate EDT (i.e., SED, 4SED$^+$, and EDT-2) all show a linear time complexity.

The 3 state-of-the-art algorithms (i.e., Maurer [6], [11], SEDT [21], and LLT* [24]), all using dimensional reduction, as well as the generic FEED class algorithm have the tendency that the execution time increases a bit with increasing size of the images. Of the 3 state-of-the-art algorithms it is proven that the number of integer arithmetic operations is indeed proportional to the number of pixels. However, with using modern computing systems this might nevertheless not translate into linear time. This depends on the efficiency of the various caches modern computing systems use [37], [38], [39]. These memory effects can be caused by the number of matrices and vectors the algorithms use and by their second phase's column wise processing. Additionally, some variance among the algorithms can be explained by characteristics of the computing systems and the varying compiler optimization parameters that or are not used, such as using pointers instead of indices.

Factors related to the computing systems' architectures and their utilization might even have a larger effect than



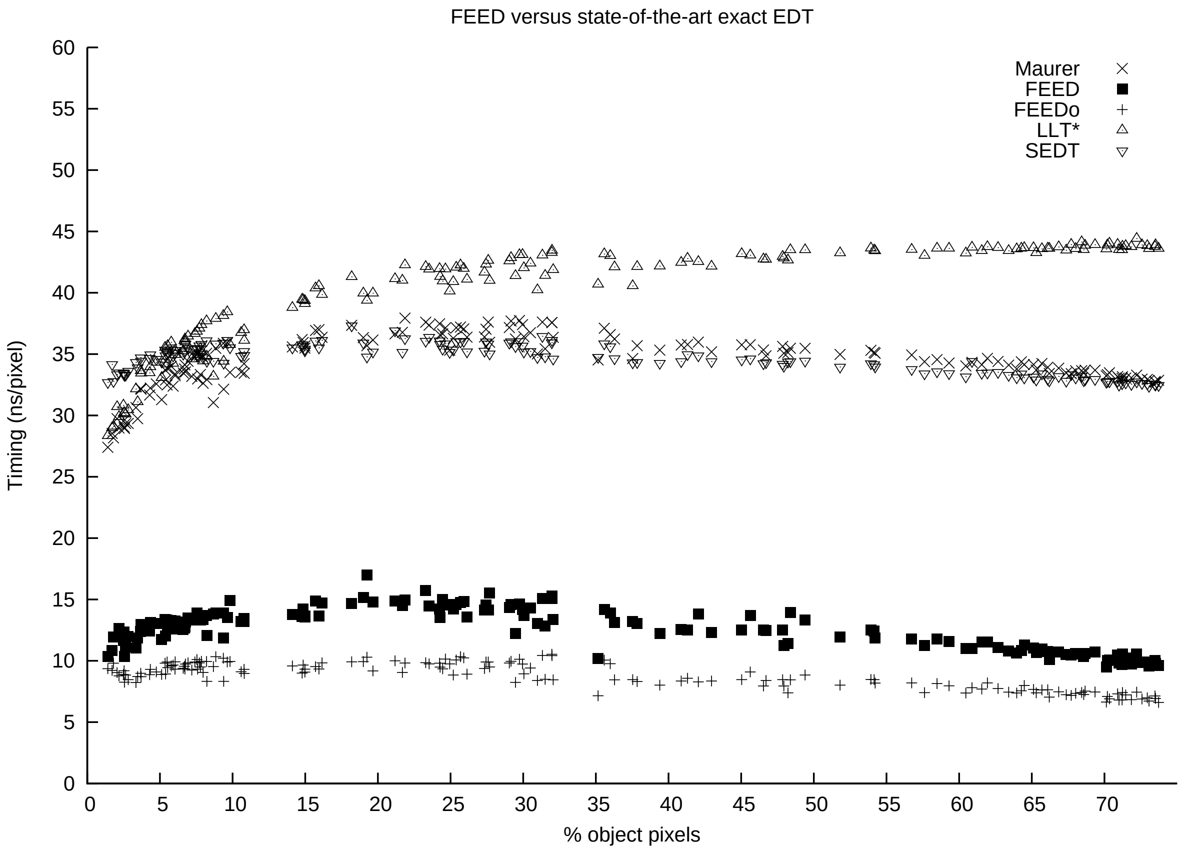


Fig. 4. The average execution time in *ns/pixel* as function of the $\%$ of object pixels in the images of the four sets of object O images described in Section 8, and illustrated in Fig. 3.

the number of arithmetic operations per pixel, which is up to now considered as the most important characteristic to minimize when developing algorithms [37], [39]. For the current benchmark, the curves of all algorithms are (nevertheless) roughly consistent with an $O(N)$ algorithmic complexity (see Fig. 5) [11]. The adapted FEED class algorithm, FEED$_o$, has a time complexity that mimics that of CH34 [13] closely. So, also FEED$_o$ has $O(N)$ algorithmic complexity (see Fig. 5).

10 Discussion

DT are a basic operation in computational geometry. In about 50 years of research on distance transforms (DT) [1], numerous algorithms have been developed [5], [7]. As such they are applied within various applications (e.g., [4], [7], [9], [10], [11], [40]), either by themselves or as intermediate method. This article introduced FEED transforms, which start from the *inverse* of the DT definition: each object pixel feeds its distance to all background pixels. The FEED class cannot be captured in the classification of existing DT algorithms: it unites features of ordered propagation, raster scanning, and independent scanning DT (see also Section 1). most importantly, FEED class algorithms have a unique property: they can be adapted to specific image characteristics.

To test FEED class algorithms, benchmarks were conducted on both the Fabbri et al. [5] data set and a novel data set consisting of object images (see Section 8), developed to reflect characteristics of realistic images (see Tables 2 and 3). The benchmarks confirmed that FEED class algorithms i) are a class of truly exact EDT; ii) outperform any other approximate or exact EDT (see also [7], [20], [25]) and its algorithmic complexity is $O(N)$; and iii) can be adapted for any image type.

DTs can result in disconnected Voronoi tiles since the tiles of the Voronoi diagram on a discrete lattice are not necessarily connected sets [4], [5], [6], [31]. This problem originates from the definition of DT [1], [2]: a DT makes an image in which the value of each pixel is its distance to the set of object pixels O in the original image. In contrast, FEED is implemented directly from the definition in 2 or rather its inverse: each object pixel O *FEEDs* its distance to all non-object pixels. Consequently, FEED (and its search strategies) does not suffer from the problem of disconnected Voronoi tiles. This indeed can be considered as yet another advantage of FEED, compared to the majority of other (E)DTs [5], [6], [31].

The 2D FEED class can be extended to 3D and even n-dimensional (nD) as was shown by the authors in [41]. In n dimensions, a border pixel B is defined as an object pixel with at least one of its neighbors with a $(n-1)$ hyperplane in common in the background. So, in 2D a border pixel B is defined as an object pixel with at least one of its four 4-connected pixels in the background. In $3D$ this becomes at least one of the six 6-connected voxels and so forth. For details, please consult [41].

In future research, instances of the FEED class will be developed to make them even faster by loosing some accuracy. This is interesting for applications where high speed is prevalent over accuracy. Moreover, with FEED it should be possible in non metrical L_p distances with $0 < p < 1$. To calculate this, [42] needed two local masks, while considering a minimum. FEED, being directly derived from L_p, would most likely need only one mask without any restriction. Also, several strategies to parallelize FEED transforms can be explored. Parallelization has already been shown to provide significant speed-ups with various other DT (e.g., [11]).

Taken together, a unequaled class of DT is introduced: the FEED class, which is fast, provides true exact EDT, does not suffer from disconnected Voronoi tiles, and can be tailored to the images under investigation. Two exhaustive benchmarks have been executed, which confirmed these claims. This quartet of proven properties marks this new class of DT as promising.

Acknowledgments

This publication was supported by both the Dutch NWO ToKeN project Eidetic (634.000.001) and the Dutch national program COMMIT. We gratefully acknowledge Andras Hajdu and Attila Fazekas (University of Debrecen, Hungary), Robin Strand and Gunilla Borgefors (Uppsala University, Sweden), and two anonymous reviewers who all provided us detailed feedback on this article. Additionally, we thank Lynn Packwood (University of Twente, The Netherlands) for her careful proofreading.

References

[1] A. Rosenfeld and J. L. Pfaltz, "Sequential operations in digital picture processing," *Journal of the ACM*, vol. 13, no. 4, pp. 471–494, 1966.

[2] ——, "Distance functions on digital pictures," *Pattern Recognition*, vol. 1, no. 1, pp. 33–61, 1968.

[3] J. Mukherjee, "On approximating Euclidean metrics by weighted t-cost distances in arbitrary dimension," *Pattern Recognition Letters*, vol. 32, no. 6, pp. 824–831, 2011.

[4] O. Cuisenaire and B. Macq, "Fast Euclidean transformation by propagation using multiple neighborhoods," *Computer Vision and Image Understanding*, vol. 76, no. 2, pp. 163–172, 1999.

[5] R. Fabbri, L. da F. Costa, J. C. Torelli, and O. M. Bruno, "2D Euclidean distance transform algorithms: A comparative survey," *ACM Computing Surveys*, vol. 40, no. 1, p. Article 2, 2008.

[6] C. R. Maurer, Jr., R. Qi, and V. Raghavan, "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, pp. 265–270, 2003.

[7] E. L. van den Broek and T. E. Schouten, "Distance transforms: Academics versus industry," *Recent Patents on Computer Science*, vol. 4, no. 1, pp. 1–15, 2011.

[8] G. Peyré, M. Péchaud, R. Keriven, and L. D. Cohen, "Geodesic methods in computer vision and graphics," *Foundations and Trends in Computer Graphics and Vision*, vol. 5, no. 3–4, pp. 197–397, 2010.

[9] J. Gao and L. Guibas, "Geometric algorithms for sensor networks," *Philosophical Transactions of the Royal Society A: Mathematical, Physical & Engineering Sciences*, vol. 370, no. 1958, pp. 27–51, 2012.

[10] Z. Zhang, H. S. Seah, C. K. Quah, and J. Sun, "GPU-accelerated real-time tracking of full-body motion with multi-layer search," *IEEE Transactions on Multimedia*, vol. 15, no. 1, pp. 106–119, 2013.

[11] K. C. Ciesielski, X. Chen, J. K. Udupa, and G. J. Grevera, "Linear time algorithms for exact distance transform," *Journal of Mathematical Imaging and Vision*, vol. 39, no. 3, pp. 193–209, 2011.

[12] G. Borgefors, "Distance transformations in arbitrary dimensions," *Computer Vision, Graphics, and Image Processing*, vol. 27, no. 3, pp. 321–345, 1984.

[13] ——, "Distance transformations in digital images," *Computer Vision, Graphics, and Image Processing*, vol. 34, no. 3, pp. 344–371, 1986.

[14] P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 227–248, 1980.

[15] Q.-Z. Ye, "The signed Euclidean distance transform and its applications," in *Proceedings of the 9th International Conference on Pattern Recognition (ICPR)*, vol. 1. Rome, Italy: Piscataway, NJ, USA: IEEE Computer Society Press, November 14–17 1988, pp. 495–499.

[16] E. Coiras, J. Santamaria, and C. Miravet, "Hexadecagonal region growing," *Pattern Recognition Letters*, vol. 19, no. 12, pp. 1111–1117, 1998.

[17] Z. Kulpa and B. Kruse, "Algortihms for circular propagation in discrete images," *Computer Vision, Graphics, and Image Processing*, vol. 24, no. 3, pp. 305–328, 1983.

[18] F. Y. Shih and Y.-T. Wu, "Fast Euclidean distance transformation in two scans using a 3×3 neighborhood," *Computer Vision and Image Understanding*, vol. 93, no. 2, pp. 195–205, 2004.

[19] ——, "The efficient algorithms for achieving Euclidean distance transformation," *IEEE Transactions on Image Processing*, vol. 13, no. 8, pp. 1078–1091, 2004.

[20] E. L. van den Broek, T. E. Schouten, P. M. F. Kisters, and H. C. Kuppens, "Weighted Distance Mapping (WDM)," in *Proceedings of the IEEE International Conference on Visual Information Engineering (VIE2005)*, N. Canagarajah, et al., Eds. Glasgow, UK: Wrightsons - Earls Barton, Northants, 2005, pp. 157–164.

[21] D. Coeurjolly and A. Montanvert, "Optimal separable algorithms to compute the reverse Euclidean distance transformation and discrete medial axis in arbitrary dimension," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 3, pp. 437–448, 2007.

[22] T. Hirata, "A unified linear-time algorithm for computing distance maps," *Information Processing Letters*, vol. 58, no. 3, pp. 129–133, 1996.

[23] A. Meijster, J. B. T. M. Roerdink, and W. H. Hesselink, *A General Algorithm for Computing Distance Transforms in Linear Time*, 2nd ed., ser. Computational Imaging and Vision. New York, NY, USA: Kluwer Academic / Plenum Publishers, 2002, vol. 18, ch. 2 (Part 8: Algorithms), pp. 331–340.

[24] Y. Lucet, "New sequential exact Euclidean distance transform algorithms based on convex analysis," *Image and Vision Computing*, vol. 27, no. 1–2, pp. 37–44, 2009.

[25] T. E. Schouten and E. L. van den Broek, "Fast Exact Euclidean Distance (FEED) Transformation," in *Proceedings of the 17th IEEE International Conference on Pattern Recognition (ICPR 2004)*, J. Kittler, M. Petrou, and M. Nixon, Eds., vol. 3, Cambridge, United Kingdom, 2004, pp. 594–597.

[26] A. Hajdu and T. Tóth, "Approximating non-metrical Minkowski distances in 2D," *Pattern Recognition Letters*, vol. 29, no. 6, pp. 813–821, 2008.

[27] M. W. Jones, J. A. Bærentzen, and M. Sramek, "3D distance fields: A survey of techniques and applications," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 4, pp. 581–599, 2006.

[28] R. Strand, B. Nagy, and G. Borgefors, "Digital distance functions on three-dimensional grids," *Theoretical Computer Science*, vol. 412, no. 15, pp. 1350–1363, 2011.

[29] M. Miyazawa, P. Zeng, N. Iso, and T. Hirata, "A systolic algorithm for Euclidean distance transform," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 7, pp. 1127–1134, 2006.

[30] G. Borgefors, I. Nyström, and G. Sanniti di Baja, "Discrete geometry for computer imagery," *Discrete Applied Mathematics*, vol. 125, no. 1, p. [special issue], 2003.

[31] O. Cuisenaire, "Locally adaptable mathematical morphology using distance transformations," *Pattern Recognition*, vol. 39, no. 3, pp. 405–416, 2006.

[32] F. Aurenhammer, "Voronoi diagrams: A survey of a fundamental geometric data structure," *ACM Computing Surveys*, vol. 23, no. 3, pp. 345–405, 1991.

[33] R. E. Sequeira and F. J. Prêteux, "Discrete Voronoi diagrams and the SKIZ operator: A dynamic algorithm," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 10, pp. 1165–1170, 1997.

[34] H. Breu, J. Gil, D. Kirkpatrick, and M. Werman, "Linear time Euclidean distance transform algorithms," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 5, pp. 529–533, 1995.

[35] R. L. Ogniewicz and O. Kübler, "Voronoi tessellation of points with integer coordinates: Time-efficient implementation and online edge-list generation," *Pattern Recognition*, vol. 28, no. 12, pp. 1839–1844, 1995.

[36] W. Guan and S. Ma, "A list-processing approach to compute Voronoi diagrams and the Euclidean distance transform," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 7, pp. 757–761, 1998.

[37] T. E. Schouten and E. L. van den Broek, "Fast multi class distance transforms for video surveillance," *Proceedings of SPIE (Real-Time Image Processing)*, vol. 6811, pp. 681 107–681 107–11, 2008.

[38] K. Moreland, "A survey of visualization pipelines," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 3, pp. 367–378, 2013.

[39] J. G. Wingbermuehle, R. K. Cytron, and R. D. Chamberlain, "Optimization of application-specific memories," *IEEE Computer Architecture Letters*, [in press].

[40] E. L. van den Broek, T. E. Schouten, and P. M. F. Kisters, "Modeling human color categorization," *Pattern Recognition Letters*, vol. 29, no. 8, pp. 1136–1144, 2008.

[41] T. E. Schouten, H. C. Kuppens, and E. L. van den Broek, "Three dimensional fast exact Euclidean distance (3D-FEED) maps," *Proceedings of SPIE (Vision Geometry XIV)*, vol. 6066, p. 60660F, 2006.

[42] A. Hajdu and L. Hajdu, "Approximating the Euclidean distance using non-periodic neighbourhood sequences," *Discrete Mathematics*, vol. 283, no. 1–3, pp. 101–111, 2004.

Theo E. Schouten has a background in experimental physics (MSc, 1970, Katholieke Universiteit Nijmegen, The Netherlands (KUN); PhD, 1976, KUN). Subsequently, he held positions at the experimental high energy physics laboratory of the KUN, the Dutch National Institute for Nuclear and High Energy Physics (NIKHEF), CERN, Switzerland, and the Max Planck Institute für Physik und Astrophysik, Germany. In 1984, he returned to the KUN (renamed to Radboud University Nijmegen) as an assistant professor in computer science, which he remained since then. He has developed, lectured, and coordinated a broad range of courses, including computer graphics, robotics, computer vision, and neural networks, as well as BSc and MSc tracks. Additionally, he guided 100+ BSc, MSc, and PhD students. Further, Theo currently serves on the editorial board of the Central European Journal of Computer Science. Theo has published 80+ scientific articles and has a patent application pending. In 2013, Theo retired from the RU but continued his research and, additionally, works as a consultant through his company theos.nl.

Egon L. van den Broek has a background in Artificial Intelligence (AI) (MSc, 2001, Radboud University Nijmegen, The Netherlands, RU). He obtained his first PhD on Content-Based Image Retrieval (CBIR) (2005) from the RU and his second PhD on Affective Signal Processing (ASP) (2011) from the University of Twente (UT), The Netherlands. Since 2004, Egon has been an assistant professor; subsequently, at the Vrije Universiteit Amsterdam, at the UT, and, currently, at the Utrecht University, all the Netherlands. He has developed and lectured several courses and MSc tracks and guided more than 50 BSc, MSc, post-doctoral, and PhD students. Additionally, he works as a consultant (e.g., for TNO, Philips, and the United Nations). Egon serves as external expert for various agencies, as PC member of conferences, on boards of advice, and on editorial boards of journals, an encyclopedia, and book series. He has published more than 150 scientific articles and has several patent applications pending. Egon frequently serves as invited and keynote speaker and has received several awards.