This chapter has been published as: A. Meijster, J. B. T. M. Roerdink and W. H. Hesselink, A general algorithm for computing distance transforms in linear time. In: *Mathematical Morphology and its Applications to Image and Signal Processing*, Kluwer Acad. Publ., 2000, pp. 331-340.

Chapter 2

A General Algorithm for Computing Distance Transforms in Linear Time

Abstract

A general algorithm for computing distance transforms of digital images is presented. The algorithm consists of two phases. Both phases consist of two scans, a forward and a backward scan. The first phase scans the image column-wise, while the second phase scans the image row-wise. Since the computation per row (column) is independent of the computation of other rows (columns), the algorithm can be easily parallelized on shared memory computers. The algorithm can be used for the computation of the exact Euclidean, Manhattan (L_1 norm), and chessboard distance (L_∞ norm) transforms.

2.1 Introduction

Distance transforms play an important role in many morphological image processing applications. They have been extensively studied and used in computational geometry, image processing, computer graphics and pattern recognition, e.g., [17, 18, 25, 81]. The two-dimensional distance transform can be described as follows. Let B be a set of grid points taken from a rectangular grid of size $m \times n$. The problem is to assign to every grid point (x,y) the distance to the nearest point in B. If we use the Euclidean metric for computing distances, and represent B by a boolean array $b[\cdot,\cdot]$, we thus want to compute the two dimensional array $dt[x,y] = \sqrt{EDT(x,y)}$, where

$$EDT(x,y) = \texttt{MIN}(i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge b[i,j] : (x-i)^2 + (y-j)^2).$$

Here we use the notation $\texttt{MIN}(k : P(k) : f(k))$ for the minimal value of $f(k)$ where k ranges over all values that satisfy $P(k)$.

Since the exact Euclidean distance transform is often regarded as too computationally intensive, several algorithms have been proposed that use some mask which is swept over the image in two scans, to compute approximations like the Manhattan (city-block) distance, the chessboard distance, or chamfer distances (see [17, 18, 25, 81]). The time complexity is linear in the number of pixels of the image (i.e. $O(m \times n)$), but it does not yield the exact Euclidean distance, which

is required for some applications. Another drawback of these algorithms is that they are hard to parallelize for execution on parallel computers since previously computed results are propagated during the computation, making the process highly sequential. A recursive algorithm of order $mn \log m$ for the exact *EDT* is given in [45]. In [75] a recursive algorithm of order mn for the exact *EDT* is given by reducing the problem to a matrix search algorithm.

In this chapter, which is based upon [54], we present an algorithm that also computes distance transforms in linear time, is simpler and more efficient than [75], and is easy to parallelize. It can compute the Euclidean (*EDT*), the Manhattan (*MDT*), and the chessboard distance (*CDT*) transform, defined by

$$\begin{aligned} EDT(x,y) &= \texttt{MIN}(i,j : 0 \le i < m \wedge 0 \le j < n \wedge b[i,j] : (x-i)^2 + (y-j)^2), \\ MDT(x,y) &= \texttt{MIN}(i,j : 0 \le i < m \wedge 0 \le j < n \wedge b[i,j] : |x-i| + |y-j|), \\ CDT(x,y) &= \texttt{MIN}(i,j : 0 \le i < m \wedge 0 \le j < n \wedge b[i,j] : |x-i| \ \mathbf{max} \ |y-j|). \end{aligned}$$

If we define the minimum of the empty set to be ∞, and use the rule $z + \infty = \infty$ for all z, we find with some calculation

$$\begin{aligned} EDT(x,y) &= \texttt{MIN}(i : 0 \le i < m : (x-i)^2 + G(i,y)^2), \\ MDT(x,y) &= \texttt{MIN}(i : 0 \le i < m : |x-i| + G(i,y)), \\ CDT(x,y) &= \texttt{MIN}(i : 0 \le i < m : |x-i| \ \mathbf{max} \ G(i,y)), \end{aligned}$$

where $G(i,y) = \texttt{MIN}(j : 0 \le j < n \wedge b[i,j] : |y-j|)$.

The algorithm can be summarized as follows. In a first phase each column C_x (defined by points (x,y) with x fixed) is separately scanned. For each point (x,y) on C_x, the distance $G(x,y)$ of (x,y) to the nearest points of $C_x \cap B$ is determined. In a second phase each row R_y (defined by points (x,y) with y fixed) is separately scanned, and for each point (x,y) on R_y the minimum of $(x-x')^2 + G(x',y)^2$ for *EDT*, $|x-x'| + G(x',y)$ for *MDT*, and $|x-x'| \ \mathbf{max} \ G(x',y)$ for *CDT* is determined, where (x',y) ranges over row R_y.

2.2 The first phase

The object of the first phase is to determine the function G. We first observe that we can split G into two functions GT (top) and GB (bottom), such that $G(i,y) = GT(i,y) \ \mathbf{min} \ GB(i,y)$, where

$$\begin{aligned} GT(i,y) &= \texttt{MIN}(j : 0 \le j \le y \wedge b[i,j] : y-j) \\ GB(i,y) &= \texttt{MIN}(j : y \le j < n \wedge b[i,j] : j-y) \end{aligned}$$

We start with the computation of GT by introducing an array g to store its values. It is easy to see that $GT(i,y) = 0$ if $b[i,y]$ holds, and that, otherwise, $GT(i,y) = \mathrm{G}T(i,y-1) + 1$ (or ∞ if $y = 0$). We can therefore compute $g[x,y] := GT(x,y)$ using only $g[x,y-1]$ in a simple column scan from top to bottom. Similarly, we find $GB(i,y) = \mathrm{G}B(i,y+1) + 1$. The second scan runs from bottom to top, and computes $G(x,y)$ directly, using GT from the previous scan, and GB from the current

```
forall x ∈ [0..m−1] do
  (* scan 1 *)
  if b[x,0] then
    g[x,0] := 0
  else
    g[x,0] := ∞;
  endif
  for y := 1 to n−1 do
    if b[x,y] then
      g[x,y] := 0
    else
      g[x,y] := 1+g[x,y−1];
    endif
  (* scan 2 *)
  for y := n−2 downto 0 do
    if g[x,y+1] < g[x,y] then
      g[x,y] := (1+g[x,y+1])
    endif
end forall
```

Figure 2.1. *Program fragment for the first phase.*

one. After some simplification, this results in the code fragment given in Fig. 2.1. Clearly, the time complexity is linear in the number of pixels (i.e. $O(m \times n)$). In actual implementations it is convenient to replace ∞ by $m+n$, since all distances in the images are less than $m+n$ if the set B is non-empty.

2.3 The second phase

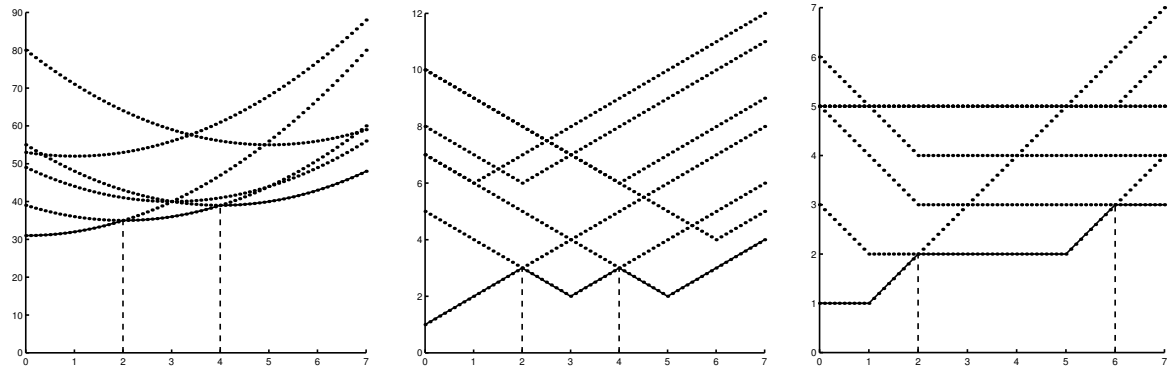
In the second phase we want to compute *EDT*, *MDT*, or *CDT* row by row, i.e. for all x with fixed y. Therefore, in this section we regard y as a constant and omit it as a parameter in auxiliary functions, and introduce $g(i) = G(i,y)$. Instead of developing an algorithm for each metric separately, we aim at a more general algorithm for

$$DT(x,y) = \texttt{MIN}(i : 0 \leq i < m : f(x,i)). \tag{2.1}$$

The choice of the function f depends on the metric we wish to use, i.e.

$$f(x,i) = \begin{cases} (x-i)^2 + g(i)^2 & \text{for } EDT, \\ |x-i| + g(i) & \text{for } MDT, \\ |x-i| \ \textbf{max}\ g(i) & \text{for } CDT. \end{cases}$$

It is helpful to introduce a geometrical interpretation of the minimization problem of Eq. 2.1. For any i with $0 \leq i < m$, denote by F_i the function $x \mapsto f(x,i)$ on the real interval $[0, m-1]$.



(a) *EDT* (b) *MDT* (c) *CDT*

Figure 2.2. DT *as the lower envelope (solid line) of curves F_i, $0 \leq i < m$ (dotted lines). The dashed vertical lines indicate the transitions between regions.*

We call i the *index* of F_i. In the case of *EDT*, the graph of F_i is a parabola with vertex at $(i, g(i))$. In the case of *MDT* the parabolas are replaced by V-shaped approximations, while in the case of *CDT* we deal with 'topped off' V-shaped approximations (see Fig. 2.2). We can interpret *DT* geometrically as the *lower envelope* of the collection $\{F_i | 0 \leq i < m\}$ evaluated at integer coordinates, cf. Fig. 2.2. The lower envelopes consist of a number of consecutive curve segments, whose index we denote by $s[0], s[1], \ldots, s[q]$ counting from left to right. The projections of the segments on the x-axis are called *regions*, and form a partition of the interval $[0, m)$ by consecutive segments. The computation of *DT* now consists of two scans. In a forward (left-to-right) scan the set of regions is determined using an incremental algorithm. In a backward (right-to-left) scan the values $DT(x,y)$ are trivially computed for all x.

We start by replacing the upper bound m in (2.1) by a variable u and define

$$FL(x,u) = \texttt{MIN}(i : 0 \leq i < u : f(x,i)).$$

The geometric interpretation is that we restrict the set B to the half plane to the left of u. Clearly, $DT(x,y) = FL(x,m)$.

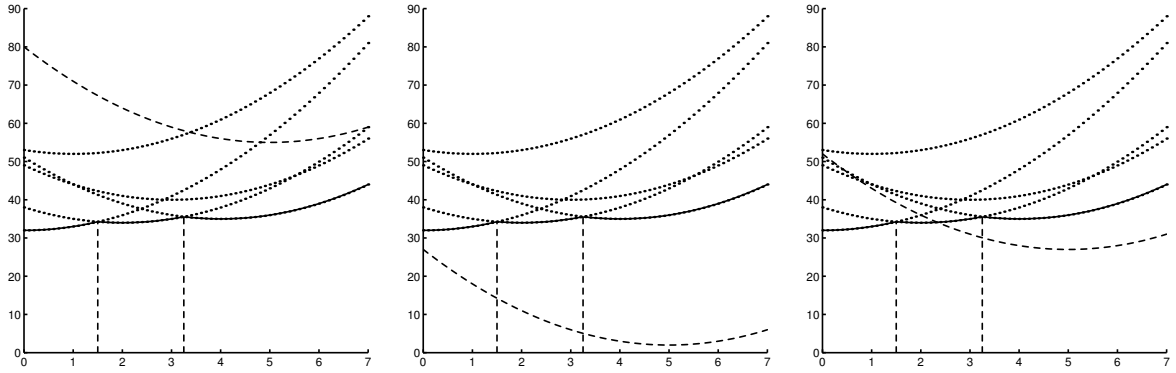
For given upper bound $u > 0$, we define an index h to be a *minimizer* at x if, in the expression for $FL(x,u)$, the minimal value of $f(x,i)$ occurs at h. In general, x may have more than one minimizer. The *least minimizer* $H(x,u)$ of x w.r.t. u is defined as the least index h with $0 \leq h < u$ such that $f(x,h) \leq f(x,i)$ for all i in the same range, i.e.

$$H(x,u) = \texttt{MIN}(h : 0 \leq h < u \wedge \forall (i : 0 \leq i < u : f(x,h) \leq f(x,i)) : h). \tag{2.2}$$

We clearly have $FL(x,u) = f(x,H(x,u))$, hence $DT(x,y) = f(x,H(x,m))$. Therefore, the problem reduces to the computation of $H(x,m)$.

We consider the sets $S(u)$ of the least minimizers that occur during the scan from left to right, and the sets $T(h,u)$ of points with the same least minimizer h. We thus define

$$\begin{aligned} S(u) &= \{H(x,u) \,|\, 0 \leq x < m\}\,, \\ T(h,u) &= \{x \,|\, 0 \leq x < m \wedge H(x,u) = h\} \text{ if } 0 \leq h < u. \end{aligned} \tag{2.3}$$



(a) above (b) below (c) intersection

Figure 2.3. *Location of F_u (dashed curve) w.r.t. the lower envelope (solid line).*

Clearly, $S(u)$ is a nonempty subset of $[0,u)$, and $S(u) = \{h \,|\, T(h,u) \neq \emptyset\}$. We define the *regions* for u to be the sets $T(h,u)$ that are nonempty. It is easy to see that the regions for u form a partition of $[0,m)$.

The aim is the case where $u = m$. Indeed, for $x \in T(h,m)$, we have $H(x,m) = h$ and hence $DT(x,y) = f(x,h)$. The second phase of the algorithm therefore consists of two scans: scan 3 computes the partition of $[0,m)$ that consists of the regions for m and scan 4 uses these regions to compute DT. For given u, only the curves with indices from 0 to $u-1$ are taken into account. The minimizer of x corresponds to the index of the curve segment whose projection on the horizontal axis contains x. Let the current lower envelope consist of $q+1$ segments, i.e. $S(u) = \{s[0], s[1], \ldots, s[q]\}$, with $s[\ell]$ the index of the ℓ-th segment. Consider what happens when F_u is added. Three situations may occur:

(a) F_u is *above* the current lower envelope on $[0,m-1]$, cf. Fig. 2.3(a). Then $S(u+1) = S(u)$, since the set $T(u,u+1)$ is empty.

(b) F_u is *below* the current lower envelope on $[0,m-1]$, cf. Fig. 2.3(b). Then $S(u+1) = \{u\}$, i.e., all old regions have disappeared, and there is one new region $T(u,u+1) = [0,m)$.

(c) F_u *intersects* the current lower envelope on $[0,m-1]$, cf. Fig. 2.3(c). The current regions will either shrink or disappear, and there is one new region $T(u,u+1)$.

We start searching from right to left for the current region which is intersected by F_u. This can be determined by comparing the values of F_u and F_ℓ at the begin point $t[\ell]$ of each current region $\ell = q, q-1, \ldots$, until we find the first $\ell = \ell^*$ such that $F_u(t[\ell^*]) \geq F_{s[\ell^*]}(t[\ell^*])$. Then F_u is not the least minimizer at $t[\ell^*]$, and there must be an intersection of F_u with F_{ℓ^*} in region ℓ^*. Let x^* be the horizontal coordinate of the intersection. If $\ell^* = q$ and $x^* \geq m$ we have case (a); if $\ell^* < 0$ we have case (b); otherwise case (c) pertains.

To find x^*, we introduce a function *Sep*, where $Sep(i,u)$ is the first integer larger or equal than the horizontal coordinate of the intersection point of F_u and F_i with $i < u$, i.e.

$$F_i(x) \leq F_u(x) \quad \Leftrightarrow \quad x \leq Sep(i,u). \tag{2.4}$$

```
forall y ∈ [0..n−1] do
  q := 0; s[0] := 0; t[0] := 0;
  for u := 1 to m−1 do (∗ scan 3 ∗)
    while q ≥ 0 ∧ f(t[q],s[q]) > f(t[q],u) do
      q := q−1;
    if q < 0 then
      q := 0; s[0] := u
    else
      w := 1+Sep(s[q],u);
      if w < m then
        q := q+1; s[q] := u; t[q] := w
      end if
    end if
  end for
  for u := m−1 downto 0 do (∗ scan 4 ∗)
    dt[u,y] := f(u,s[q]);
    if u = t[q] then q := q−1
  end for
end forall
```

Figure 2.4. *Program fragments for the second phase.*

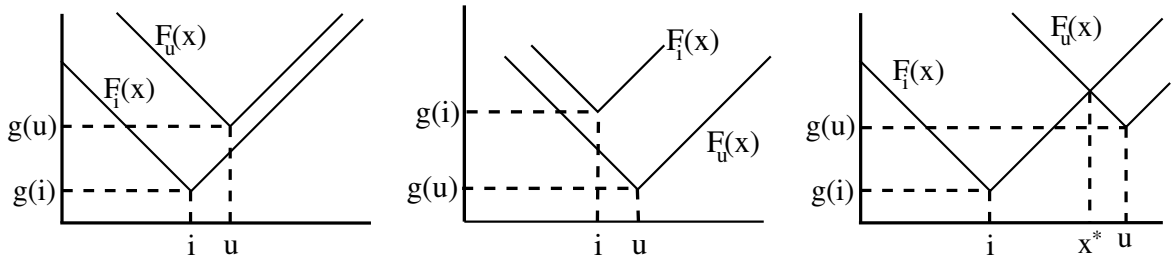
We thus have $x^* = Sep(s[l^*],u)$. Clearly, the function *Sep* is dependent on which distance transform we want to compute. In the next section we will derive the expressions for the function *Sep*, but in the remainder of this section we simply assume that *Sep* is available.

We introduce an integer program variable u. It is convenient to represent $S(u)$ by an increasing sequence of elements. Since the regions form a partition of $[0,m)$ by consecutive segments, we can represent them by the sequence of their least elements. According to the case analysis above, the regions are to be adapted at their end. We can therefore implement these sequences in two integer arrays, s and t, with an integer variable q as index of the end point.

We start with the forward scan, see scan 3 in Fig. 2.4. We have $S(1) = \{0\}$, and $T(0,1) = [0,m)$, and thus start with $q = 0$, $s[0] = 0$, and $t[0] = 0$. In a loop, variable u is incremented, and thus the representations of S and T must be updated by means of the case analysis above.

To investigate the complexity of the forward scan, we consider the expression $q + 2(m-u)$, which is initially $2m$. In every execution of the body of the outer loop (scan 3 in Fig. 2.4), and also in every execution of the body of its inner loop, the value of the expression decreases. This implies that the time complexity of the scan is linear in m. Note that, the *average* number of iterations of the inner loop is at most two. The algorithm uses less than $2m$ comparisons of f values, and function *Sep* is evaluated less than m times.

When the forward scan is finished, we have completely determined the partition of $[0,m)$ in regions. Given these regions, we can trivially compute dt-values in a simple backward scan (see scan 4 in Fig. 2.4).



(a) $g(u) \geq g(i)+u-i$ (b) $g(i) > g(u)+u-i$ (c) otherwise

Figure 2.5. *Cases for finding Sep for MDT.*

2.4 Derivation of the function Sep

The derivation in the previous section was independent of the actual metric used. The functions dependent on the metric are f and *Sep*. In this section we compute expressions for *Sep* for *EDT*, *MDT*, and *CDT*. The easiest is *EDT*. We find for $i < u$

$$\begin{aligned}
& F_i(x) \leq F_u(x) \\
\Leftrightarrow & \{\text{definition of } F_i, F_u\} \\
& (x-i)^2 + g(i)^2 \leq (x-u)^2 + g(u)^2 \\
\Leftrightarrow & \{\text{calculus; } i < u\text{; } x \text{ is an integer}\} \\
& x \leq (u^2 - i^2 + g(u)^2 - g(i)^2) \textbf{ div } (2(u-i)).
\end{aligned}$$

Here, we denote integer division with rounding off towards zero by **div**. Thus, we find for *EDT* that

$$Sep(i,u) = (u^2 - i^2 + g(u)^2 - g(i)^2) \textbf{ div } (2(u-i)).$$

If we use the Manhattan metric, the analysis is slightly more complicated. Since we have to deal with absolute values in the expressions, awkward case analysis is necessary if we want to compute *Sep* analytically. Therefore we prefer a geometric argument. We have to consider three cases (see Fig. 2.5).

If $g(u) \geq g(i)+u-i$, the graph of F_u lies entirely above the graph of F_i for all x, thus we choose $Sep(i,u) = \infty$. If $g(i) > g(u)+u-i$, the graph of F_i lies entirely above the graph of F_u, so $F_i(x) \leq F_u(x)$ for no x at all. Thus, we must choose $Sep(i,u) = -\infty$ to satisfy (2.4). In all other cases, F_u intersects F_i at $x^* = (g(u) - g(i) + u + i)/2$. So, if we want to compute *MDT* we use

$$Sep(i,u) = \begin{cases} \infty & \text{if } g(u) \geq g(i)+u-i, \\ -\infty & \text{if } g(i) > g(u)+u-i, \\ (g(u) - g(i) + u + i) \textbf{ div } 2 & \text{otherwise.} \end{cases}$$

For the case of *CDT* we have $|x-i| \textbf{ max } g(i) \leq |x-u| \textbf{ max } g(u)$. We consider two main cases, which each can be split up in three sub-cases. First we consider the case $g(i) \leq g(u)$. From Fig. 2.6(a)-(c), we see that the increasing segment of F_i ($y = x-i$) intersects the decreasing part of F_u ($y = u-x$), or the constant part ($y = g(u)$). Let γ be the vertical coordinate corresponding

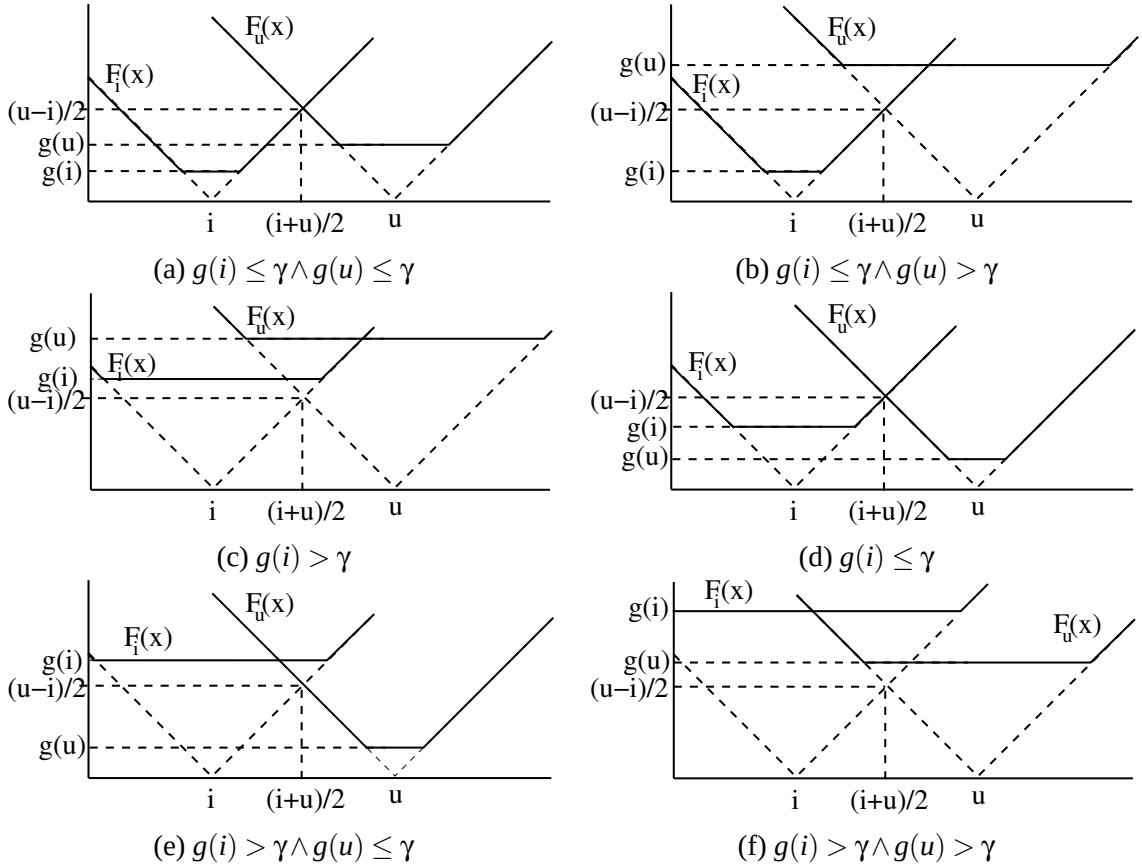


Figure 2.6. *Cases for finding Sep for CDT, where* $\gamma = (u-i)/2$. *Cases (a)-(c):* $g(i) \leq g(u)$. *Cases (d)-(f):* $g(i) > g(u)$.

with the middle of i and u ($x = (i+u)/2$), i.e. $\gamma = (u-i)/2$. From Fig. 2.6(a), we see that if $g(i) \leq \gamma \wedge g(u) \leq \gamma$, we have $F_i(x) \leq F_u(x)$ if $x \leq (i+u)/2$. From Fig. 2.6(b)-(c), we see that the increasing part of F_i intersects the constant segment of F_u at $i+g(u)$, and thus we have $F_i(x) \leq F_u(x)$ if $x \leq i+g(u)$. Putting the three cases together, we can conclude

$$g(i) \leq g(u) \Rightarrow \big(F_i(x) \leq F_u(x) \Leftrightarrow x \leq \frac{i+u}{2} \ \mathbf{max}\ (i+g(u))\big).$$

The other main case is $g(i) > g(u)$. Again, in Fig. 2.6(d), we see that if $g(i) \leq \gamma$, the intersection at $(i+u)/2$ is the separator. If $g(i) > \gamma$ (see Fig. 2.6(e)-(f)), the horizontal segment of F_i intersects the decreasing part of F_u at $x = u - g(i)$. Just like in the previous case, we can put these cases together. This results in the following expression for *Sep*:

$$Sep(i,u) = \begin{cases} (i+g(u)) \ \mathbf{max}\ ((i+u) \ \mathbf{div}\ 2) & \text{if } g(i) \leq g(u), \\ (u-g(i)) \ \mathbf{min}\ ((i+u) \ \mathbf{div}\ 2) & \text{otherwise.} \end{cases}$$

Table 2.1. *Timing results in ms. From left to right: EDT, MDT, and CDT.*

size	p=1	p=2	p=3	p=4	p=1	p=2	p=3	p=4	p=1	p=2	p=3	p=4
256	12	7	5	4	11	6	4	3	12	6	4	3
512	69	35	25	19	63	34	24	17	67	35	25	18
1024	307	156	104	79	281	147	97	74	298	152	101	77
2048	1542	780	517	389	1407	709	476	357	1501	753	506	381
4096	6251	3137	2098	1577	5753	2886	1929	1451	6073	3053	2041	1530

2.5 Parallelization, timing results and conclusions

Since the computation per row (column) is independent of the computation on any other row (column), the algorithm is well suited for parallelization on a shared memory machine. In the first (second) phase, the columns (rows) are distributed over the processors. The two phases must be separated by a *barrier*, which assures that all processors have completed the first phase before any of them starts with the second phase. The theoretical time complexity of the parallel algorithm for p processors (where $p \leq m$ **min** n) is $O(mn/p)$.

We ran experiments on an Intel Pentium III based shared memory parallel computer with 4 cpu's, running at a 550MHz clock frequency. We performed time measurements using several binary images, and found that the execution time is almost independent of image content, and scales well w.r.t. the number of processors. This is as expected, since the amount of work per row and column is almost the same. In table 2.1 the timings for square images are given for $p = 1$ to $p = 4$ processors. Note that the computation of *MDT* and *CDT* is only slightly faster than the exact *EDT*. We also implemented the sequential algorithm of [81] for *CDT*, and found that our algorithm is less than a factor of 2 slower, which can easily be overcome by parallel processing.

The algorithm can be easily extended to d-dimensional distance transforms by separating the problem into d phases, each solving a one-dimensional problem, as carried out above for the case $d = 2$.

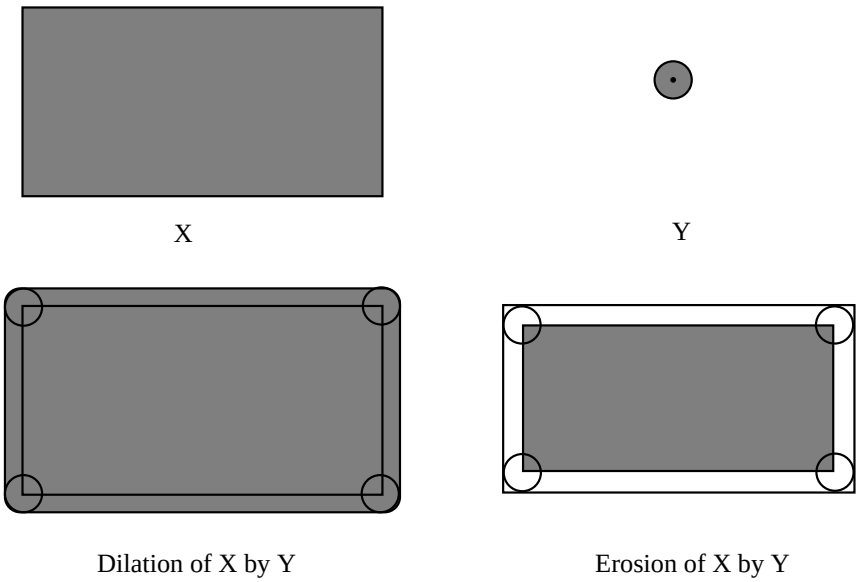


Figure 1.1. *The geometrical interpretation of dilation and erosion (in the continuous case).*

In order to avoid having to write many parentheses that make expressions hard to read, we introduce the notation

$$\overset{\vee}{Y}_h = (\overset{\vee}{Y})_h = \{h - y \,|\, y \in Y\}.$$

Dilation and erosion have a simple geometrical interpretation, which helps to understand their behavior (see figure 1.1):

$$\begin{aligned} \delta_Y(X) &= \{h \,|\, X \cap \overset{\vee}{Y}_h \neq \emptyset\} \\ \varepsilon_Y(X) &= \{h \,|\, Y_h \subseteq X\} \end{aligned}$$

In words, the dilation of X by Y is the set of points h such that the translation of the reflected set $\overset{\vee}{Y}$ over the vector h 'hits' the set X. Similarly, the erosion of X by Y is the set of points h such that after translating Y over the vector h it 'fits' in X.

Several algebraic properties of dilation and erosion exist. The most important properties are:

- Dilation and erosion are *increasing operators*, which means that if $X \subseteq Y$ we have $\delta_B(X) \subseteq \delta_B(Y)$, and $\varepsilon_B(X) \subseteq \varepsilon_B(Y)$ for any structuring element B.
- Erosion is *decreasing* in its second argument (the structuring element). If $B_1 \subseteq B_2$, then $\varepsilon_{B_2}(X) \subseteq \varepsilon_{B_1}(X)$, for every set X.
- Erosion and dilation form an *adjunction*, i.e. $\delta_B(A) \subseteq C \Leftrightarrow A \subseteq \varepsilon_B(C)$, for any sets A, B and C.
- Dilation is commutative: $\delta_B(A) = \delta_A(B)$, or equivalently $A \oplus B = B \oplus A$, for any sets A and B.