Foreword

For some reason, the foreword is often one of the last parts of a book to be written. This book is no exception. Shouldn't a *fore*word be written first, *before* the book? In that case, I should have written this in 1988, when my supervisor, Prof. Danielsson, suggested a possible variation on the Euclidean Distance Transform (EDT). But instead, I write this after the rest of the thesis, so it is rather my *after*word.

What most of this thesis is about is fast computation of distances in digital images, i.e. the EDT, either as a separate operation or as part of other operations, and either on conventional computers or multi-processor computers.

This book is a continuation of my licentiate thesis, which consists of the three first papers in this thesis. To that I have added seven more, two of these written in cooperation with others. While the first papers cover only the EDT operation, the new papers widen the scope somewhat, applying the EDT to other problems: erosion and dilation, edge smoothing, shape representation and thinning.

Since all papers are written as stand-alone papers, there is often a significant overlap, especially in the introductions, and some are written in UK english while others are written in US english, depending on my publishing intentions. The advantage with this solution is that the published papers are reprinted in versions identical or very close to the published versions (though with a common formatting). After all, the published papers, especially the journal papers, are more widely available references than a PhD thesis.

In addition to the ten papers, there is also an introductory chapter, surveying the development, algorithms and applications of the EDT and related operations.

Acknowledgements

In a desperate attempt to be original, I wanted *not* to thank each and every living thing I've met during my work, but rather to say why I don't have to. This proved to be difficult.

Most of all, it would be hypocrisy not to thank my supervisor, Professor Per-Erik Danielsson. He held most of the graduate courses I attended, put me on the track that got me started, and has given a lot of good, constructive criticism on my work. Few people

can assess a problem so fast. Thanks for helping me get started when I couldn't, for letting me not write what I wouldn't, and for telling me not to write what I shouldn't.

Many authors thank their wives, mostly for not complaining more than they did and for not locking them out after working too late. My wife, MSc Eva L. Ragnemalm, has been much more than supportive, done proofreading (her english is much better than mine), discussed wild ideas and given me valuable feedback despite working in a completely different field of research. (I don't understand anything of *her* work myself.) I definitely do owe her thanks, though I am planning to return the favours when she gets closer to her own PhD.

Other people that I just cannot avoid thanking, no matter how original I want to be, include the people I have cooperated with, especially Dr. Gunilla Borgefors, Dr. Gabriella Sanniti di Baja and Prof. Sergey Ablameyko.

It is, however, far easier to avoid thanking STU (the Swedish board for technical development), who funded much of my research. Not because I am ungrateful, but because STU no longer exists. R.I.P.

Thanking colleagues, family and friends is a difficult task, and I could waste lots of space telling how wonderful and supportive you all are – but at least I can save that space. You will get a party, isn't that good enough?

Finally, there are some people that I really must thank. During my work, I have often asked myself if the problems I have been working with are not too trivial for a PhD, but fortunately, other authors have shown up that are even more naive than I am, making fouled-up solutions to problems more trivial than the ones I work with, thereby restoring my self-confidence. I do owe these authors[1] thanks for involuntarily supporting my work.

Ingemar Ragnemalm, Linköping 1993

1. Proposition: The reader is not one of the naive authors mentioned above.

Proof: The authors in question clearly do not even read the papers they are referring to, not even the abstracts. Thus, it is not likely that they will ever care to read this thesis, much less the acknowledgements or, close to impossible, a footnote in the acknowledgements. Q.E.D.

Contents

Ten little papers

Ten little papers, my thesis would be fine,
but one was too hard to do, so I'll make do with nine.
Nine little papers, now had I made - but wait!
That is nothing new, he said and they were eight.
Eight little papers, I thought I'd make eleven.
Then I found an article that voided one, left seven.
Seven little papers were full of all my tricks.
One was way too clear, so it must go and they were six.
Six little papers, I still had left alive.
One is just an intro, so let us call them five.
Five little papers, I tried to make one more
but that linked together two old, and they were four
Four little papers, were all 'bout EDT
but then there was a hardware error, they were only three
Three little papers are left, what shall I do?
On one I'm just third author, and they were two.
Two little papers, look now, I said, I'm done.
This one isn't interesting, he said, and I had one.
One little paper was all that was left for me,
but that is most definitely not a PhD.

Written late one night in 1992 when this book felt very far away.

The Euclidean Distance Transform

Euclid (in Greek *Eukleides*, in Swedish *Euklides*) was a greek mathematician, born about 300 B.C. He probably received his early training from the pupils of Plato, and later founded a school in Alexandria during the reign of Ptolemy I, who reigned from 306 to 283 B.C. Euclid's greatest work is the *Elements* (*Elementa*), 13 books covering all the known mathematical knowledge of the time [29].

In the Elements, we find Euclid's famous postulate 5, the parallel postulate:

If a straight line falling on two straight lines makes interior angles on the same side with a sum less than two right angles, the two straight lines, if produced indefinitely, meet on that side on which the angle sum is less than two right angles [41].

This postulate can not be proven. Rather, it is valid in, and defines, the *Euclidean space*, which is not the only space that can be described mathematically. The Euclidean space is a space consisting of all ordered sets $(x_1,\ldots, x_n)$ of n numbers with the distance between $(x_1,\ldots, x_n)$ and $(y_1,\ldots, y_n)$ being given by

$$\left[\sum_{i=1}^{n}(x_i-y_i)^2\right]^{\frac{1}{2}}$$

The number n is called the dimension of the space [27]. This distance is what we in the following call the *Euclidean distance*.

1. Introduction

In the (at least seemingly) continuous physical world, we generally use the *Euclidean distance* metric when measuring distance. The Euclidean distance is the rotation invariant measure that we generally would consider the "real" distance.

In image processing, however, our images consist of arrays of discrete pixels. In this discrete world, the Euclidean distance is much less practical. As a simple example, take the game of chess. The chessboard, with its squares, is like the discrete image. The distance from one corner to any other corner is 7 steps for a king, even though the Euclidean distance along a side is much shorter than over the diagonal. This is because the chess king uses the *Chessboard distance*.

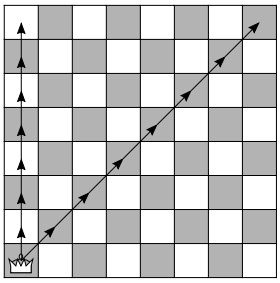


Figure 1. For the king, the distance along the side is the same as along the diagonal.

For the designer of image processing algorithms, the discrete image is somewhat like a chessboard but with a lot more squares, and for him it is much easier to use a simple metric like Chessboard distance for many tasks. Another equally simple metric is the City Block distance, which is the distance we get if we are only allowed to travel horizontally or vertically, like a chess rook.

Let us take a simple example. Suppose that we have a binary (black-and-white) image. Then, for some reason, we want to expand (dilate) the black objects, make them larger. Let us say, we want expansion with 4 pixel distances. A simple way to do that is to change all white pixels that have a black neighbour to black, and repeat that operation 4 times. If we consider all pixel positions in a 3·3 square to be neighbours, then this procedure will expand the shapes by 4 Chessboard distances, as illustrated by Figure 2.

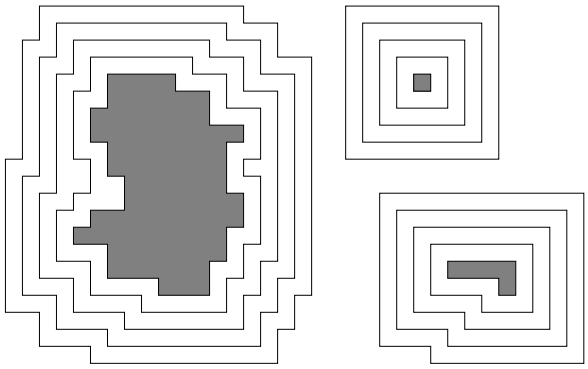


Figure 2. Some shapes being expanded 4 pixel distances.

Another way to do it, using the Euclidean distance, is the following. For each white pixel, test all pixels within the chosen distance, measured as the crow flies. If any of these pixels is black, the white pixel is set to black. The result is shown in Figure 3.

Unfortunately, this is a brute force solution of the problem, which is computationally very inefficient. Imagine that the expansion distance is large, of the same order as the N pixel units along one side of the whole image. Then, this test requires that we examine $O(N^2)$ pixels for each pixel which is a candidate for change. Since there are $O(N^2)$ such candidates, the procedure has a complexity in $O(N^4)$. The procedure illustrated in Figure 2 is more efficient. To expand O(N) steps requires O(N) scans of the image. Thus, the complexity of that operation is $O(N^3)$.

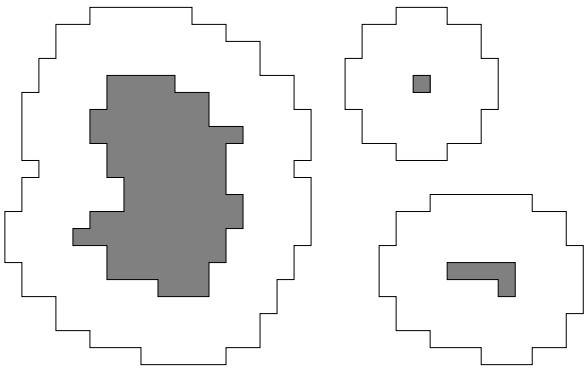


Figure 3. Expansion by finding all pixels within a certain distance from each and every black pixel.

What this thesis will be about is to solve both these problems at once, to design algorithms that combine high precision with speed. Actually, we will see that rotation invariant expansion, i.e. with Euclidean metric, is possible to obtain with algorithms that have complexity in $O(N^2)$. The expansion problem above is just one out of many image processing tasks that can be addressed with the algorithms presented.

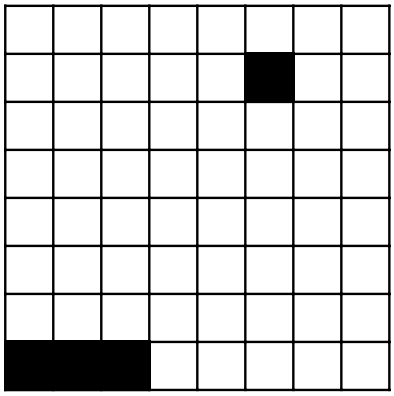
2. The Distance Transform

2.1 Definition

In image processing, we usually deal with discrete images, images that are sampled into discrete pixels. A *binary* image is an image where the pixels can only have two values, 0 or 1, black or white, true or false, foreground or background, feature or non-feature. A binary image is generally the result of a thresholding operation, classifying all pixels in the image into these two classes.

Then, if we are interested in analyzing shape properties of a binary image, a useful tool is the *Distance Transform* (DT). The operation takes a binary image as input and generates a greyscale image as output, the *Distance Map* (DM), in which each pixel holds the distance to the closest pixel in a chosen domain. See Figure 4. In the Figure, all pixels in the resulting distance map holds the distance (in the City Block distance metric) to the closest black pixel in the input image.

The Euclidean Distance Transform (EDT), producing Euclidean distance maps, is a special case that we will return to in Section 3.3.



6	5	4	3	2	1	2	3
5	4	3	2	1	0	1	2
5	5	4	3	2	1	2	3
4	4	4	4	3	2	3	4
3	3	3	4	4	3	4	5
2	2	2	3	4	4	5	6
1	1	1	2	3	4	5	6
0	0	0	1	2	3	4	5

Figure 4. A binary image and its distance map (City Block distance).

Throughout this thesis, we will generate distance maps sometimes in the *background* and sometimes in the *foreground*. The algorithms are the same, simply swapping the two domains, so we make no effort to keep them apart. In the following, we will generally consider the distance maps to hold distances to *feature pixels*, which may be foreground or background as appropriate.

It should also be noted that the terms *distance transform* and *distance transformation* are often used for the output as well as the operation. I have chosen to, as far as possible, use the term *distance map* for the output, in order to make the distinction more clear. A common synonym for distance map, most common in older papers, is *distance function*. We sometimes also find the term *distance mapping* as a synonym for distance transform.

2.2 Variations on the Distance Transform

There are some important variations on the distance transform concept, which I will briefly introduce in this section.

An important special case of the DT is the *constrained distance transform* (CDT), where the source image consists not only of feature and non-feature pixels, but also of *obstacles*. A distance value in a constrained distance map shows the distance to the closest object pixel, not along a straight line, but along a path that does not pass through any obstacle pixels. This kind of DT is useful for path planning problems, e.g. in robot applications. A simple example is shown in Figure 5.

There is an even more special case of the CTD, namely the *distance transformation of line patterns* (DTLP), useful for certain line pattern analysis tasks. This is essentially a CDT where all paths must have 1-pixel width.

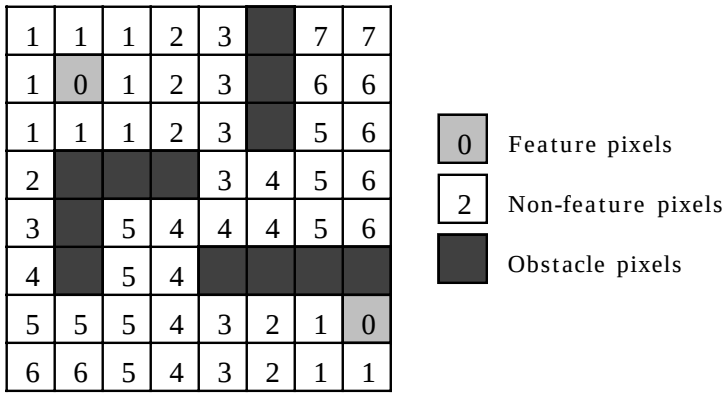


Figure 5. Constrained distance transformation.

Another variation on the DT concept is *labelled* distance transforms, producing labelled distance maps. In this case, the distance map does not only hold a distance value, but also information about the originating pixel, either denoting a subclass of the feature pixels or a specific feature pixel. The Signed Euclidean Distance transform (see further below) is a labelled distance transform since it, for each pixel, holds a pointer to the originating feature pixel.

3. The development of distance transformation algorithms

3.1 Reducing computation time

If we disregard the computation time, the problem of generating distance maps is trivial. In such a case, we can use a brute force algorithm that for every pixel in the binary image searches all the feature pixels in order to find the closest one, and this thesis would end right here.

However, such an algorithm is extremely inefficient. For a 2-dimensional image with n·n pixels, the processing time would be in $O(n^4)$, assuming the number of feature pixels grow with the image size. With current computers, it would take minutes or even hours to process an ordinary image, for a problem that, as we shall see, can be solved in less than a second!

If computation time is of any importance at all (which it usually is), the problem of generating a distance map, that is, to design a distance transform, becomes much harder. All fast algorithms have one thing in common: for each pixel processed, we do operations involving only very few other pixels, generally in some neighbourhood of the pixel being processed.

The first distance transforms using neighbourhood operations were parallel algorithms. With parallel, we mean that the same operation is applied to all pixels in the image at once, producing a new image. What is calculated in one position in the image can not affect any other parts until the next iteration. Such algorithms are very suitable for parallel computers, but less so for ordinary sequential (single processor) computers.

According to the references of Rosenfeld and Pfalz [62], the first papers about DTs discussed parallel DTs. Unfortunately, these papers are from the early 60's and are hard to acquire today. A parallel DT, producing a City Block distance map (and that is probably similar to the early parallel DTs) works as follows:

*

Let d(x,y) be the distance value at location (x,y). Initialize the distance map to 0 for all feature pixels and to N in all non-feature pixels (where N is a value that is bigger than the largest possible distance value).

In each iteration, do the following operation for every pixel in the image. Let the pixel being processed (the *center pixel*) be located at coordinates (x, y). For each 4-neighbour (the north, east, west and south neighbouring pixels), get the distance values and add 1. Assign the center pixel the minimum value of these four values and the distance value of the center pixel itself. This can be written:

$$d(x,y) := \min\{ d(x,y), d(x+1,y)+1, d(x-1,y)+1, d(x,y+1)+1, d(x,y-1)+1 \}$$

Let the algorithm iterate until there are no changes in the image during one iteration.

*

This algorithm is much faster than the brute force algorithm. Instead of, for every pixel in the image, visiting all pixels or at least all feature pixels, we only have to visit the 4-neigbors a number of times proportional to the largest distance in the distance map. This distance can be expected to be proportional to the image size, which means that the resulting computational complexity is in $O(n^3)$ for a 2-dimensional image with $n \cdot n$ pixels.

In most recent papers, and throughout this thesis, it is common to visualize DTs with a set of masks, illustrating what neighbourhoods that are used in each stage, and showing what distance offset is used for each neighbour. For the parallel algorithm just described, the mask, in this case the *4-neighbourhood*, is shown in Figure 6.

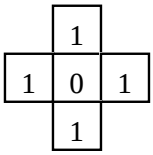


Figure 6. The mask for a parallel DT.

Still, it is possible to make even faster algorithms. The first DT that was suitable for sequential (single processor) computers was proposed by Rosenfeld and Pfalz [62] in 1966, and has been the model for a lot of later work, including several of the papers in this thesis. This DT scans the image twice, row by row, once from top to bottom and once from bottom to top. For each pixel visited, its distance value is compared to the distance value plus 1 of each of two neighbours, and the lowest value is written in the pixel being processed. Rosenfeld's algorithm can be written in pseudo code as follows (adapted from [62]). The image is initialized as the parallel algorithm above.

```
for y:=0 to N do
      for x:=0 to N do
            d(x,y) := min{ d(x,y), d(x-1,y)+1, d(x,y-1)+1 }
for y:=N downto 0 do
      for x:=N downto 0 do
            d(x,y) := min{ d(x,y), d(x+1,y)+1, d(x,y+1)+1 }
```

This is a recursive, *raster scanning* algorithm. We can also illustrate it with a set of masks, see Figure 7. Compare these masks with the algorithm written above. Note that what has happened is essentially that we have cut the mask for the parallel algorithm (Figure 6) in half, and apply one half in the first scan and the other half in the second scan.

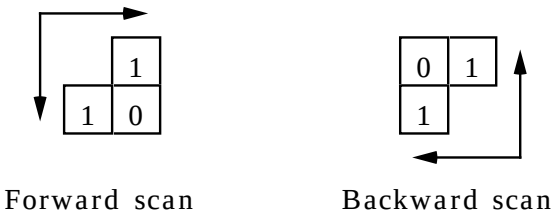


Figure 7. The masks for a 2-scan DT.

3.2 Improving precision

Both the two algorithms described above have a common drawback. They produce DMs with the *City Block distance* metric. This metric gives us highly rotation dependent results. For example, the two shapes in Figure 8 have approximately the same shape and size (squares), but one is rotated 45˚. The distance values are far from equal in corresponding positions, e.g. the center.

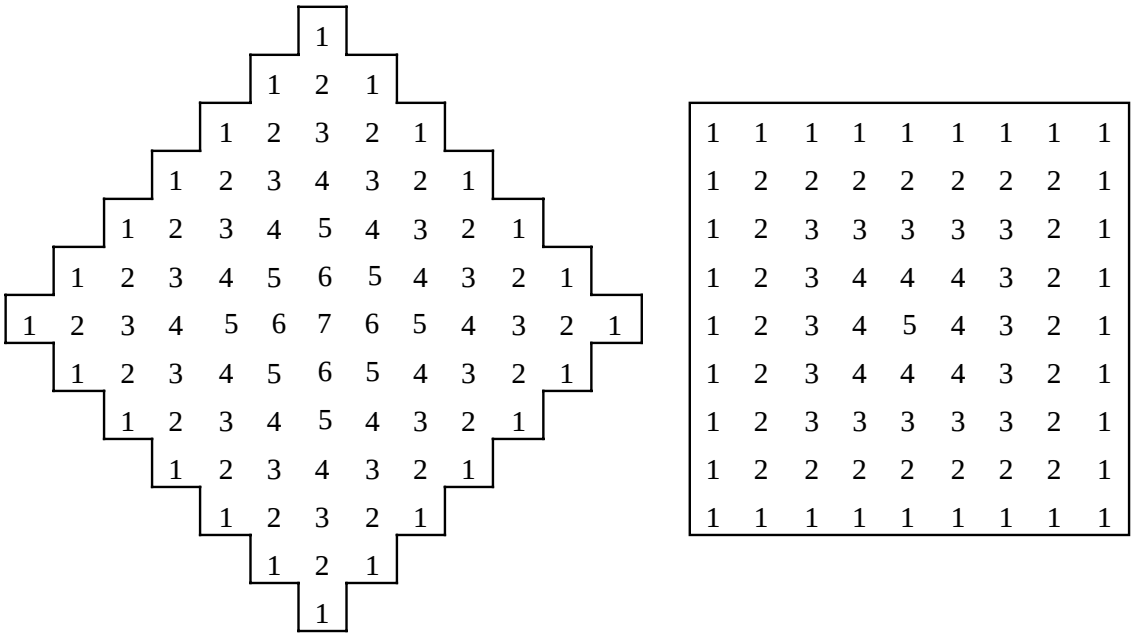


Figure 8. Distance maps in two squares with City Block distance .

Another way to demonstrate the precision, or lack thereof, of a metric is to study its equidistance curve. Such a curve consists of all points on a given distance from a point. For Euclidean distance, the equidistance curve is a circle. For the City Block distance, it is a square. See Figure 9.

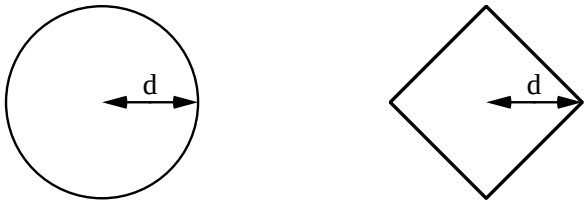


Figure 9. The shape of equidistance curves for Euclidean versus City Block distance (i.e. disks of radius d).

Hence, there was an urge to find alternative neighbourhoods that could create DMs with higher precision. One solution is to abandon the normal Cartesian, rectangular grid and switch to the hexagonal grid (see Figure 10). This will not only give us better approximations of the Euclidean distance, but also solve the connectivity problem, the problem that the foreground and background must have different connectivity when using d^8 or d^4 connectivity in the Cartesian grid (the question whether the diagonal neighbours are considered neighbours). In the hexagonal grid, this problem disappears. However, there are other problems, especially the lack of input and output devices for images with this kind of sampling.

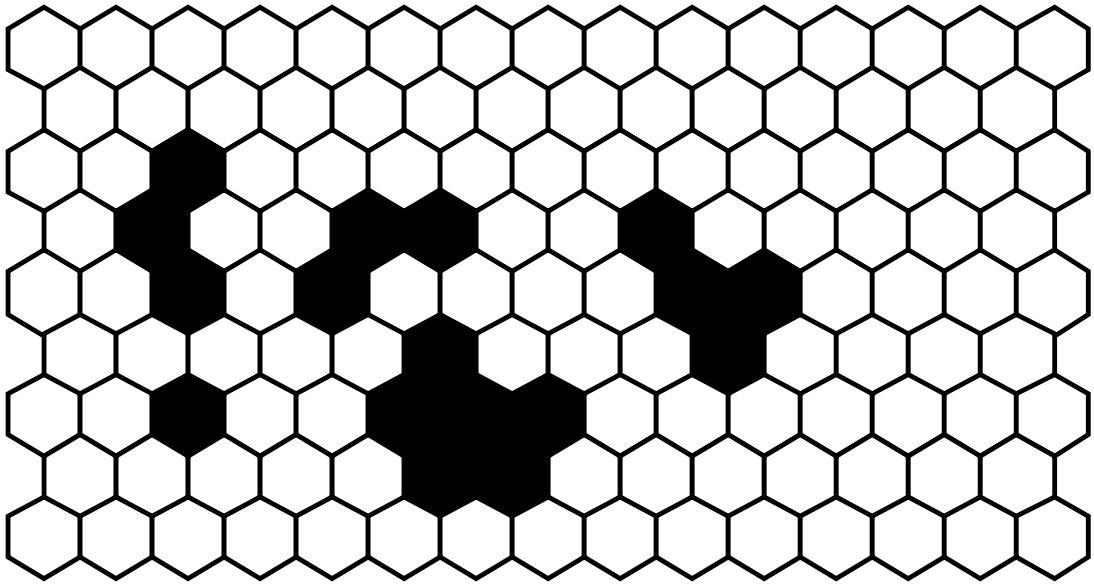


Figure 10. A hexagonally sampled image.

In the hexagonal grid, the simplest metric is the hexagonal distance, also known as honeycomb distance, using a 6-pixel neighbourhood with offset 1 to every pixel. Luczak and Rosenfeld did early work with this metric [39]. The complete mask is shown in Figure 11. Borgefors [17] has investigated the possibilities for further improvements of DTs in hexagonally sampled images.

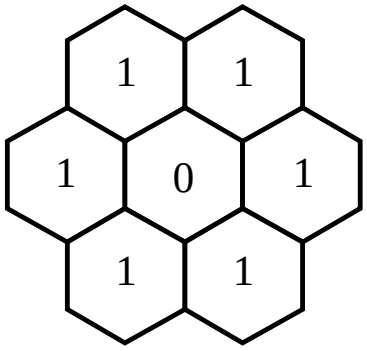


Figure 11. Complete mask for honeycomb distance.

Another method that has often been used to get higher precision with simple algorithms is to use the *octagonal* metric. The idea is simple. If we use City Block distance, as above, we get correct distances in horizontal and vertical directions from the feature pixels, but too small distance values in diagonal directions.

The Chessboard metric, however, uses a full 3·3 neighbourhood, with distance 1 in all directions. See Figure 12, which shows the parallel mask (left) and the raster scanning masks (right). With this metric, we get too large distance values in diagonal directions rather than too small.

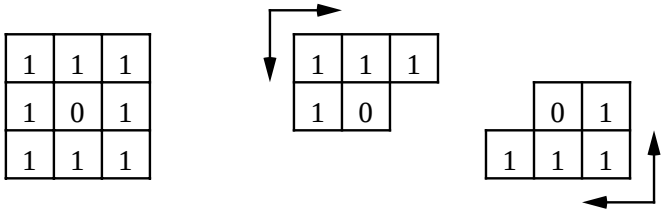


Figure 12. The masks for Chessboard distance DT, parallel and raster scanning.

By alternating these two, we get a much better approximation to Euclidean distance, comparable to the 3·3 weighted DTs (see below), but using only integers with offsets of 1. Alternating them using one step City Block for each step of Chessboard has been quite popular. I call this Octagonal 1:1 distance. More recently, other combinations have been proposed [25,26], e.g. using one step of Chessboard for each two steps City Block, which I choose to call Octagonal 2:1 distance.

However, it is far from trivial to make a sequential octagonal DT algorithm. The simple splitting into two masks, as is done for other DTs above, is not possible. This problem was addressed by Danielsson [22], but he also suggested a much better solution, namely the Euclidean Distance Transform, which we will describe in Section 3.3.

Very early, Montanari [43] gave us another, very flexible solution, namely *weighted* distance transforms, using Quasi-Euclidean metrics. He proposed that rather than using the distance 1 for all neighbours, we could use different values in different directions. The most obvious mask is probably the one shown in Figure 13, with distance 1 in horizontal and vertical directions, and $\sqrt{2}$ in diagonal directions.

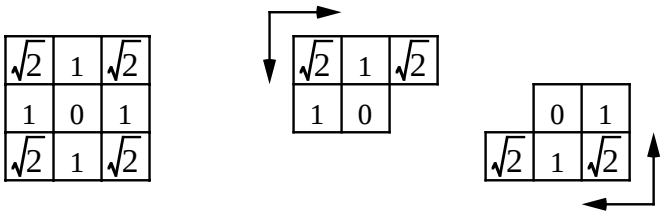


Figure 13. Masks for a weighted DT, parallel and raster scanning.

Figure 13 shows the parallel mask and the two sequential masks we get by splitting the parallel one into two scans, like we did for the City Block distance DT described above. The masks should obviously give perfect results in the eight basic directions, but the distances in other directions will not be exact. The equidistance curves will be polygons, as illustrated in Figure 14.

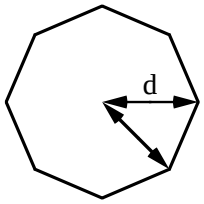


Figure 14. The equidistance curve in a weighted DT is a polygon.

If we allow scaling, as proposed by Vossepoel [72], the DT in Figure 13 is the optimal 3·3 mask for a weighted DT. This is fairly obvious since its equidistance curve is a symmetric octagon and therefore the optimal approximation of a circle with an 8-sided polygon. Vossepoel uses the name quasi-Euclidean distance for this optimal case. The main drawback with this solution is that it uses real numbers rather than integers. This drawback caused the weighted DT technique to be less useful until the development of integer-based algorithms, as proposed by Barrow et. al. [7] and developed further by Borgefors [14,21], Vossepoel [72] and Beckers and Smeulders [9]. 3-D versions have been proposed by Borgefors [16] and more recently by Beckers and Smeulders [8].

With integer-based weighted DTs, we use masks like in Figure 13, but using integer numbers larger than the actual distance. This means that the distance values generated are scaled up, but can be scaled down after the distance transform has been completed. In the 3·3 neighbourhood, the 2-3 Chamfer DT [72] and the 3-4 Chamfer DT [14,16] are popular choices. For the 5·5 neighbourhood, Borgefors [14] recommends the Chamfer 5-7-11 DT. The complete masks for these DTs are shown in Figure 15. Each mask can be split in two in order to get a raster-scanning algorithm.

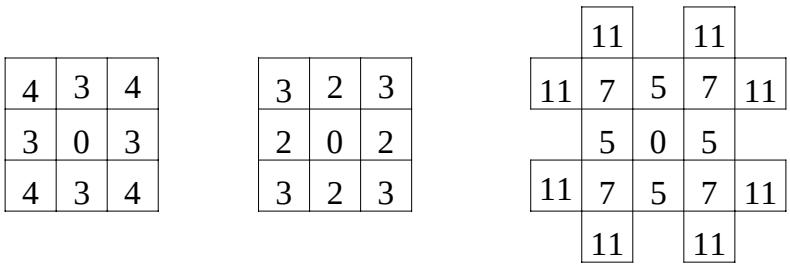


Figure 15. Masks for Chamfer 3-4, Chamfer 2-3 and Chamfer 5-7-11 DTs (parallel masks).

With bigger neighbourhoods, we can get higher precision. This is simply because the equidistance curves are now polygons of higher order (in the 5·5 case, a 16-sided polygon) which can approximate a circle better than an 8-sided polygon. Figure 16 shows the equidistance contours of some weighted DTs (including the non-integer one from Figure 13) as well as a number of other DTs.

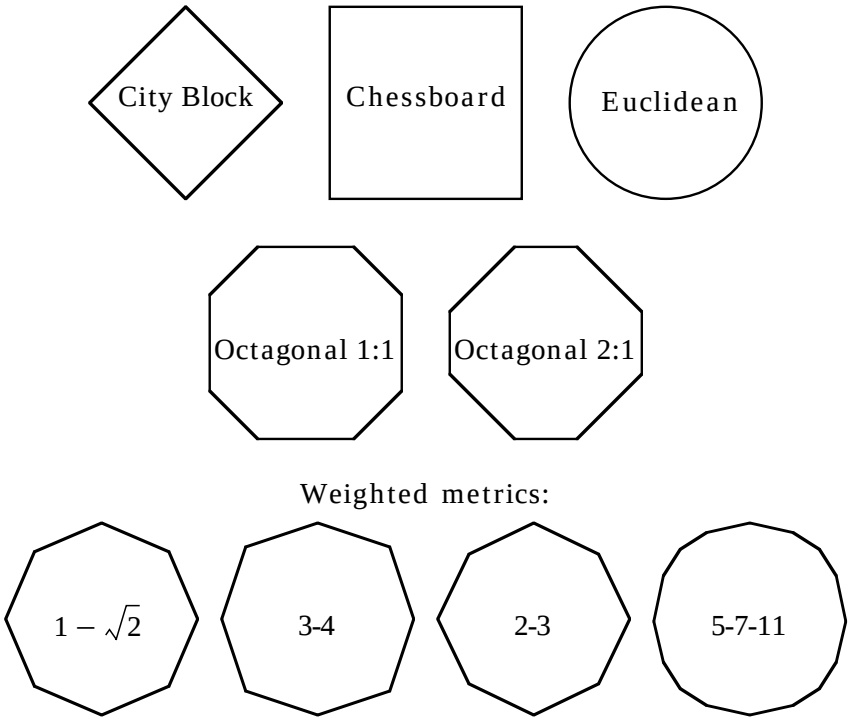


Figure 16. The equidistance contours for 9 different metrics.

As we can see in Figure 16, the 5-7-11 DT approximate the Euclidean DT very well. However, this is at the expense of a considerably slower algorithm.

Let us make a note about the neighbourhoods used for weighted DTs. With bigger neighbourhoods, masks consisting of the complete neighbourhoods (e.g. 5·5 or 7·7, see Figure 17) are no longer necessarily the best choice. We can weed out some pixels, thereby making the algorithm run faster.

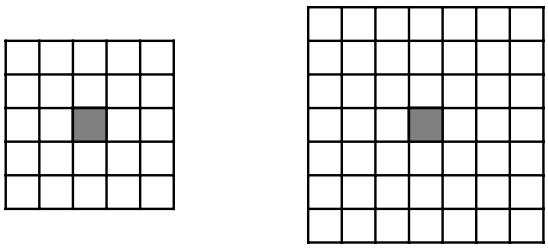


Figure 17. Complete 5·5 and 7·7 neighbourhoods. Center pixels are shaded.

We now define the concept of *prime vectors*, which has some importance for the following discussions. This concept was introduced in my licentiate thesis [50].

Definition: A *prime vector* is a vector with integer components whose components have a largest common divisor of 1.
Definition: A *non-prime vector* is any vector with integer components that is not a prime vector, that is, a vector whose components have a largest common divisor larger than 1.

This means that any non-prime vector can be expressed as some prime vector multiplied by an integer, and no prime vector can be expressed in such a way other than as itself times 1. Thus, in a weighted DT, a mask member on non-prime vector from the center pixel will not be necessary for the propagation since any propagation it produces can always be produced as several steps of propagation with the mask member that is on the corresponding prime vector. Hence, we can remove all non-prime vectors from the mask and still produce the same distance map.

This assumes that the distance values on the non-prime vector positions are scaled from the distance value of the prime vector. If this is not true, one of these mask members will not contribute at all, which makes it meaningless to include anyway.

When making weighted DTs with larger masks than 3·3, it is common practice to use this result, thereby using *sparse masks*, using only the prime vectors. For 5·5 and 7·7 neighbourhoods, this results in the masks in Figure 18.

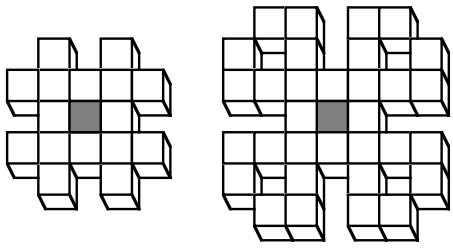


Figure 18. A 5·5 and a 7·7 neighbourhood with unnecessary mask members removed, that is, sparse masks. The picture has 3D-perspective to make holes in the masks visible.

It is theoretically interesting to investigate how many pixels can be removed for each mask size for larger masks. This is illustrated by Figure 19, which shows the prime vectors for masks up to a radius of 10.

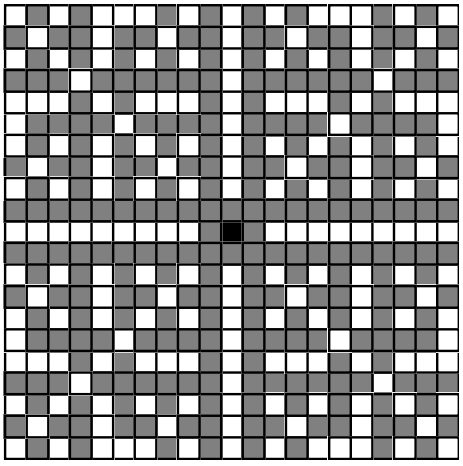


Figure 19. The shortest prime vectors, shown as a grid of pixels. The black pixel is origin, grey pixels are on prime vector positions and white are on non-prime vector positions.

By simply counting the pixels, we get the following table, where n is the side of the mask and p is the number of prime vectors in the mask, center pixel not counted. $(p+1)/n^2$ is the ratio of all vs. necessary mask members.

n	**p**	**$(p+1)/n^2$**
3	8	1.00
5	16	0.68
7	32	0.67
9	48	0.60
11	80	0.67
13	96	0.57
15	144	0.64
17	176	0.61
19	224	0.62
21	256	0.58

This table goes far beyond realistic mask sizes. We may conclude that we get a reduction of the computation for each mask by 30-40% for realistic mask sizes.

For parallel algorithms, there is another option that is more interesting, namely *shell masks*. With shell masks, parallel algorithms can be several times faster than using sparse masks.

For sparse masks, we removed all non-prime vectors. This causes different propagation speed in different directions, and most notably reduces parallel propagation to 1 pixel distance per iteration. This is because the propagation speed in some direction is proportional to the length of the prime vector in that direction.

We may use the following masks instead: for each set of vectors $a\cdot\mathbf{v}$, a = 1, 2, 3...k within the mask, keep only the vector $k\cdot\mathbf{v}$. All others are removed. This mask, the *shell mask*, is then used in all iterations except the first one, where a complete mask is used. A shell mask for a 7·7 neighbourhood is shown in Figure 20.

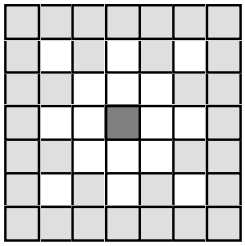


Figure 20. 7·7 shell mask. The shaded pixels are included in the mask.

For the 7·7 neighbourhood, this improves the speed by a factor of 3, performing 3 steps of propagation per iteration rather than one, with the same processing time per iteration (slightly more in the first iteration).

In my licentiate thesis [50], several other variants on DT masks were outlined. However, for raster scanning algorithms, the sparse masks are all we need for a weighted DT.

3.3 The Euclidean Distance Transform

Danielsson [22,24] proposed that rather than propagating scalar values, be it integers or real values, we could use vectors. If each pixel is assigned a vector referring to the closest feature pixel, it is possible to generate distance maps where prefect results are achieved in all directions. This was the first published *Euclidean Distance Transform* (EDT).

Danielsson proposed three different EDT algorithms, the 4SED (4-point Sequential Euclidean Distance mapping), 8SED (8-point Sequential Euclidean Distance mapping) and PED (Parallel Euclidean Distance mapping).

8SED works as follows:

We use an image where each pixel holds a 2-component vector $v_{ij} = (v_x, v_y)$. Vectors in feature pixels are initiated to (0,0), while vectors in non-feature pixels are initiated to (Z,Z), where Z is a sufficiently big number.

Then each line from top to bottom is scanned back and forth, and each pixel visited is tested against a set of neighbours:

```
for j := 1 to N do
    begin
        for i := 1 to N do
        for i := N downto 1 do
```

$$v_{ij} := \min \begin{cases} v_{ij} \\ v_{i+1.j} + (1.0) \end{cases}$$

```
    end
```

$$v_{ij} := \min \begin{cases} v_{ij} \\ v_{i-1.j} + (1.0) \\ v_{i-1.j-1} + (1.1) \\ v_{i,j-1} + (0.1) \\ v_{i+1,j-1} + (1.1) \end{cases}$$

The min operation selects the vector with the smallest length. Also note that all image border checks are omitted in order to make the description simpler. Then, all rows are scanned back and forth again, this time from bottom to top:

```
for j := N downto 1 do
    begin
        for i := 1 to N do
```

$$v_{ij} := \min \begin{cases} v_{ij} \\ v_{i+1.j} + (1.0) \\ v_{i-1.j+1} + (1.1) \\ v_{i,j+1} + (0.1) \\ v_{i+1,j+1} + (1.1) \end{cases}$$

```
        for i := N downto 1 do
```

$$v_{ij} := \min \begin{cases} v_{ij} \\ v_{i-1.j} + (1.0) \end{cases}$$

```
    end
```

We can express this with masks as shown in Figure 21. The numbers in the figure show the vector to be added to the center pixel.

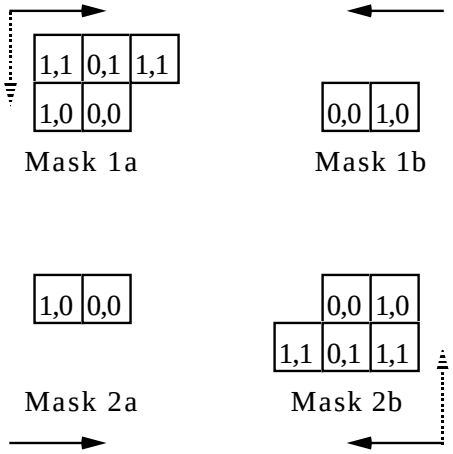


Figure 21. Masks for 8SED, an unsigned Euclidean distance transform.

The PED algorithm is a parallel EDT that uses four simple propagation primitives, shown in Figure 22. Each of the four masks is applied for all pixels at once (in parallel),. After applying all four, one step of propagation corresponding to applying a complete 3·3 mask is done.

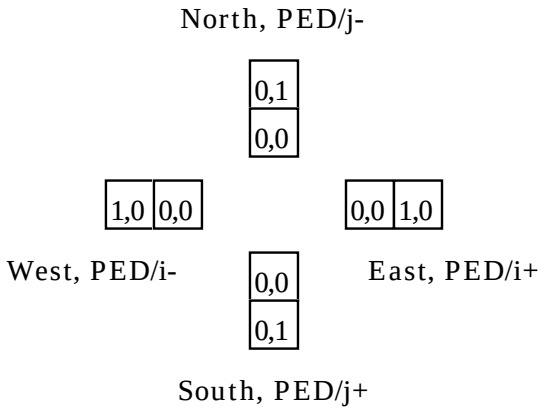


Figure 22. The four masks for PED, a parallel EDT.

In these suggested forms of EDT, only positive vector components were used, though the possibility of using signed vectors was also pointed out. Ye [75] developed the signed EDT further, with signed versions of the above algorithms. The most interesting version is probably the 8-pixel version, the 8SSED (8-pixel Signed Sequential Euclidean Distance mapping). Its masks are shown in Figure 23. Note that the only difference to 8SED is that we can have negative signs on components in the offset vectors.

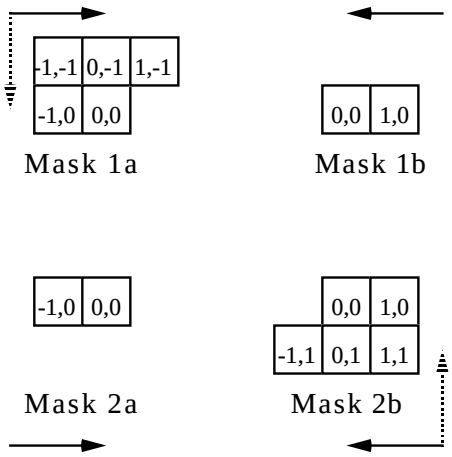


Figure 23. Masks for 8SSED, a signed Euclidean distance transform.

Ye [75] demonstrates several applications for the signed EDT. Earlier, the same year as Danielssons first papers, Fischler and Barrett [30] had presented applications for a similar algorithm, using signed vectors. These two papers demonstrate how powerful the new technique is. The usefulness of the signed version is so much bigger that we will hardly consider the unsigned version through the rest of this thesis. It is no more complex than the unsigned version, adding only a sign bit per vector component.

As pointed out already by Danielsson [22], the 4SED and 8SED, as well as their signed versions 4SSED and 8SSED, are not totally error-free. This will be discussed in more depth in Section 4.

Finally, let us note that the vector-based approach to EDT is useful even in the hexagonal grid (in the case where we use hexagonal grid for avoiding the problem with connectivity, as noted in Section 3.2, rather than for using honeycomb distance). Luczak and Rosenfeld [39] use a coordinate system with three non-orthogonal coordinate axes in the 2D plane. Let us use only two axes, x and w. Any vector in the plane can still be represented with components along these axes, like $\bar{L} = x\hat{x} + w\hat{w}$. See Figure 24.

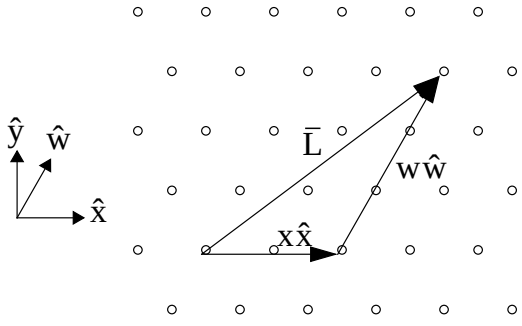


Figure 24. The vector $\bar{L}$ expressed in the components x and w.

The base vector $\hat{w}$ can be expressed in the common, orthogonal Cartesian coordinates as

$$\hat{w} = w_x\hat{x} + w_y\hat{y} = \cos\frac{\pi}{3}\hat{x} + \sin\frac{\pi}{3}\hat{y}$$

Thus, $\bar{L}$ in Cartesian coordinates is:

$$\bar{L} = x\hat{x} + y\hat{y} = \left(x + \frac{w}{2}\right)\hat{x} + \frac{\sqrt{3}}{2}w\hat{w}$$

This gives us the length of the vector in Euclidean distance:

$$\|\bar{L}\| = \sqrt{\left(x + \frac{w}{2}\right)^2 + \frac{3w^2}{4}} = \sqrt{x^2 + xw + w^2}$$

Thus, it is very simple to calculate the Euclidean distance from the (x,w)-coordinates. Moreover, the squared distance is, just like in the Cartesian grid, always integer. This allows us to use exclusively integer arithmetics through the calculation of the distance map, only using square roots in a final step, where we can use lookup tables. Vincent [70] has proposed an EDT on the hexagonal grid, using the equation above to calculate distance.

4. Precision

In this section, I will discuss the precision of various DTs, i.e. how well they approximate the Euclidean distance. Especially, I will discuss how well various versions of EDT perform in this respect.

As pointed out by Danielsson [22], the first EDTs were not totally error-free. To understand why this is the case, the reader should be familiar with the concept of Voronoi Diagrams [31]. For each feature pixel in the image, we can find a polygon that encloses all the pixels that are closer to that feature pixel than to any other feature pixel. In a correct distance map, the pixels inside the polygon should hold the distance to that feature pixel.

Such a polygon is a Voronoi polygon, and the union of all Voronoi polygons form the Voronoi Diagram of the image. See Figure 25.

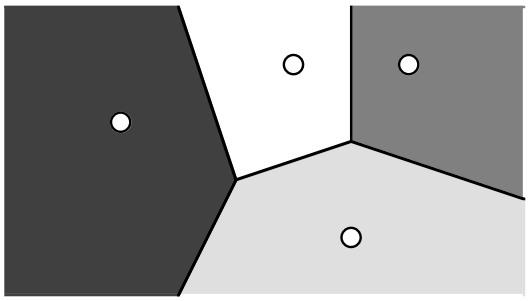


Figure 25. A Voronoi diagram from four points.

For some arrangements of feature pixels, the corner of a Voronoi polygon can have a very acute angle. That can give us a corner that is so narrow that the propagation in the distance transform can not be guaranteed to reach the outmost pixels. In such a case, these pixels can get a value that does not refer to the closest feature pixel, but to one slightly farther away. See Figure 26.

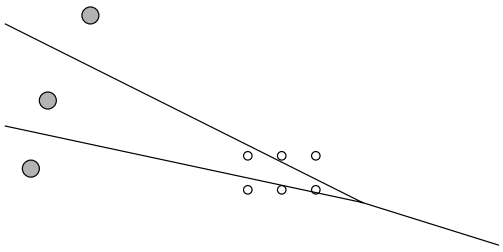


Figure 26. A feature pixel configuration where the pixel in the corner is not guaranteed to be reached by the propagation from the middle feature pixel.

In the EDT algorithms, errors occur only in this kind of corners with acute angles, which are uncommon in most types of images. Also, when the errors occur, they are very small. Danielsson [22] reports the maximum error in 8SED to be 0.090 pixel distances, and for 4SED a maximum error of 0.29 pixel distances. For practical problems, these small, infrequent deviations are without any significance.

It should be stressed that these errors only occur in sequential algorithms. Yamada [74] has proven that a parallel EDT with 3·3 neighbourhood is error-free.

Since the original EDT algorithms, 4SED and 8SED, are not completely error-free, some authors have lumped these algorithms together with non-Euclidean algorithms using City Block, Chessboard, octagonal and weighted metrics. Here, I would like to make a distinction between errors of different types.

With a *non-Euclidean* DT, we refer to a metric where the distance value assigned to each pixel is *generally* not a Euclidean distance value. This is true for all the simple or weighted metrics. Second, the maximum error, the maximum deviation from the Euclidean distance, has *no upper bound* for these DTs, but grows linearly with increasing distance.

In what we call a *Euclidean* DT, every pixel has a vector and thereby a distance value that gives the exact Euclidean distance to *some* feature pixel. Even if this is not in every case the closest feature pixel, the distance is still Euclidean and rotation invariant. In the following, we will call all algorithms EDTs if this condition is fulfilled. Admittedly, it is a weak condition, and not sufficient to make any EDT interesting.

Most EDTs, however, also have an absolute upper bound on the error. For 8SED, it is as small as 0.09 pixel distances, as mentioned previously. Taking sampling noise into account (from which an error of at least 0.5 pixel distances should be expected), it is inappropriate to put the 8SED in the same bag as the truly non-Euclidean City Block, Chessboard and weighted DT algorithms, that have no upper bound on the error.

More specifically, I propose the following classification by precision of Distance Transforms:

A: Error-free DTs.

In this group, we find the trivial but impractical brute force algorithms, most parallel EDT's and some sequential EDT algorithms. However, as noted in the next section, no sequential error-free algorithm proposed so far can quite match the algorithms in class B in computational speed (including the one I propose in paper #2). One algorithm, due to Rutowitz [63], runs in $O(N^2)$ for any image, while the other algorithms all have worst cases where they are slower.

B: DTs with an absolute upper bound on errors.

All DTs in this group are Euclidean, including Danielsson's 4SED and 8SED, Ye's 8SSED, 3-dimensional raster scanning algorithms proposed by Borgefors [16] and Mohr and Bajcsy [42] and some new algorithms proposed in this thesis (papers #1,2,7). If the Euclidean distances in an EDM are rounded to integers, preferably up, as proposed by Rhodes [60], the result is also an EDM in this class.

C: No absolute upper bound on errors.

This group range from the poor City Block and Chessboard DTs to the very good weighted DTs. It also includes some rather odd EDTs, where the upper bound on errors has been

sacrificed for speed. It is generally possible to find an upper bound for the *relative* error for any DT in this group.

Disregarding brute force algorithms, the first EDT that was shown to be error-free was Yamada's parallel EDT [74], in 1984.

In 1987, in a paper mainly concerned with thinning, Klein and Kübler [33] briefly outlined an EDT that is claimed to be error-free, but that is very computationally intensive. Their description is very brief, but the algorithm appears to be nothing but an optimized brute force algorithm.

In 1989, two much faster algorithms were presented. Rutowitz [63] proposed an algorithm for generating Voronoi diagrams as well as the EDT. The algorithm scans the image from top to bottom and then from bottom to top. For each row, a list, the active set, holds references to all feature pixels whose Voronoi polygons reach that row. This list is updated each time a new feature pixel is found during the scanning process. The algorithm can be proven to run in $O(N^2)$ for an $N \cdot N$ image, though this is with a relatively high constant due to calculations of Voronoi polygon corners [64]. This does not necessarily make it the fastest error-free EDT, but the most robust one concerning speed.

At the very same conference [52], I presented my error-free EDT (paper #2) based on Yamada's results. It uses *ordered propagation* (called *contour processing* in my earlier papers) which emulates the propagation process of the parallel algorithm, but processing only the propagation front in each iteration. It buffers all updates in order to perfectly emulate the updating process in a parallel EDT. Hence, Yamada's proof [74] is valid for this algorithm as well, which proves this new algorithm to be error-free.

For the hexagonal grid, Vincent [70] proposed an EDT in 1991 that uses *chain propagation*, that is, ordered propagation using a chain-coded propagation front. By allowing several propagation fronts to exist on top of each other, queues can exist that give the same effect as the ones appearing when using Yamada's [74] and my own algorithms [52]. Note that in the Cartesian grid this queuing process can take place in the image, while in the hexagonal grid, where Yamada's proof is not valid, it must be explicitly supported by the structures representing the propagation front.

More recently, in 1992, Paglieroni [48] has proposed a so called *unified* DT. By unified, he refers to that the DTs produced are *labelled* (see page 11), and can therefore use any metric desired, a general property of all vector-based DTs like the Signed EDT that was recognized much earlier by Fischler and Barrett [30]. Paglieroni's algorithm separates horizontal and vertical scanning, which has some advantage for parallel implementations. First, a horizontal labelled DT is done on each row, a trivial operation. Unfortunately, the column processing involves testing an unknown number of pixels for each pixel, which makes it easy to find worst cases for this algorithm where it is substantially slower than conventional EDTs.

Also in 1992, Mullikin [44] proposed yet another error-free EDT that works in 3D as well as 2D. In this case, a scheme is set up to collect all ties and near-ties, which occurs on and near the borders of the Voronoi polygons. This is done on a pixel by pixel basis. Mullikin shows that by only collecting the ties, where two feature pixels are on exactly

the same distance to a pixel, the number of errors is reduced substantially for a not too big computational cost. By collecting and processing near-ties up to a difference of 1 pixel distance, a completely error-free EDT is achieved, but for a significant computational cost. Thus, while this algorithm is of limited interest in its error-free form, the version collecting ties has interest both as a near error-free EDT and, more importantly, for use in algorithms where arbitrary assignment of ties is not acceptable.

Finally, to illustrate the multitude of solutions to this problem, I will suggest yet another method, a post-processing step for the EDT. As noted above, the errors of a raster scanning EDT will only occur in corners of Voronoi polygons with acute angles. It is possible to find these corners, thereby finding pixels that have not been reached by the propagation from the appropriate pixel. This can be done with the following procedure:

Using an appropriate neighbourhood for detecting corners (e.g. a mask pair with 1·3 and 3·1), search for pixels with two neighbours in other Voronoi polygons, that is, with vectors pointing to two feature pixels different from the center pixel. In Figure 27, an example with a 1·3 detector mask is shown.

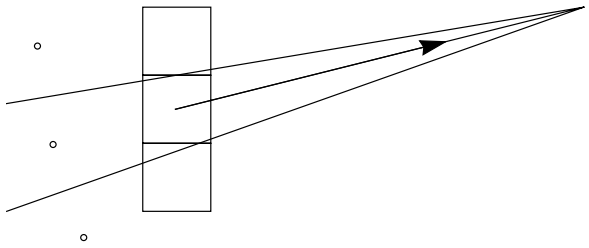


Figure 27. Shooting Stars - an outline for yet another error-free EDT.

Since the detector can give us vectors to all the three feature pixels (to the left in Figure 27), we can calculate the point where the Voronoi edges meet. If there are any pixels that should refer to the center feature pixel that have not been reached by EDT, they will be located between the detected pixel and the corner. We can then search the pixels between the detected pixel and the corner, choosing pixels with an ordinary line drawing algorithm (along the arrow in Figure 27).

This outline is obviously too simple to make a useful algorithm. Most of all, the detection of starting points for the checking must be made in a way that will not give as many false alarms. I will make no attempts to optimize the algorithm, since it will in any event be too computationally expensive to be useful in practice compared to the better algorithms above. Considering the way this algorithm works, and the insignificance of the errors it is correcting, I call it *Shooting Stars*.

Out of all the error-free algorithms listed above, Rutowitz' algorithm [63] appears to be the most attractive one, being relatively simple and not too dependent on the image data (the only error-free one always having computation cost in $O(N^2)$). My own algorithm [paper #2, 52,53] accesses fewer neighbours per pixel than any of the others, making it fast in normal cases ($O(N^2)$) but with worst cases where it gets noticeably slower.

Let us now leave the error-free algorithms and look at the other end of the precision space. As proposed in my licentiate thesis [50] and in the appendix to paper #1, we can sacrifice the inherent precision in EDT for speed, i.e. we can design a *Quick-And-Dirty SSED*. In paper #1, I state that a raster scanning EDT must support propagation in all possible directions in order to produce a correct EDT, which guarantees that the EDT will be at least in group B. If we use Quick-And-Dirty SSED, we choose not to include all directions, but instead reducing the number of scans and the number of pixels in each mask in order to improve speed. Such an algorithm will then belong to group C.

5. Computational complexity

In this section, I will return to the question of computational complexity. Since the whole distance transform problem is a question of processing speed, the computational complexity of the different algorithms is obviously of critical importance. In addition to the complexity, speed is also dependent on the optimization of the inner loops of the algorithms, a subject discussed for the EDT by Leymarie and Levine [37] and in paper #2 of this thesis [53].

In order to be able to compare parallel and sequential algorithms, we use the computational cost [2], which is the computation time times the number of processors being used.

We must also make some assumptions concerning the contents of the images being processed, since many algorithms have data-dependent processing time. We assume the following:

• The number of feature pixels is proportional to the number of pixels in the image.

• The largest distance in the resulting distance map is proportional to the side of the image.

• The length of all edges (number of black pixels with white neighbours or vice versa) is proportional to the perimeter of the image.

Thus, for a 2-dimensional image with N·N pixels, we expect the number of feature pixels to be in $O(N^2)$ and the largest distance as well as edge length in O(N).

For a 3-dimensional image, size N·N·N, we expect a number of feature pixels in $O(N^3)$ but a largest distance in O(N). In 3D, we should consider surfaces rather than edges, which are 2-dimensional, so we can expect the number of surface pixels to be in $O(N^2)$.

The assumptions made about the image data may, of course, be different if we are working with some specific application where we have more a priori knowledge. Mullikin [44] works with microscopy images where there are reasons to assume thin objects. In that case, the number of feature pixels will be of the same order as the number of edge pixels. However, I believe that the assumptions made above are appropriate when no a priori knowledge exists.

I propose the following main classes of algorithms according to their computational complexity (ordered in decreasing complexity):

A: Brute force

2D: $O(N^4)$
3D: $O(N^6)$
The class of brute force algorithms include the most trivial brute force algorithm, where for all feature pixels, all pixels are tested, as well as some better ones where all feature pixels are tested for every pixel, and some even more elaborate methods. Generally speaking, algorithms in this class are not interesting for practical use.

B: Parallel

2D: $O(N^3)$
3D: $O(N^4)$
With a parallel algorithm, we do not necessarily refer to an algorithm that is executed on a parallel architecture. Rather, we refer to algorithms that are computed in a manner comparable to a parallel algorithm, typically where we process all pixels in one iteration in order to get a fixed number of propagation steps (typically one).

As discussed in paper #1, normal parallel algorithms are not cost effective compared with sequential algorithms implemented in a parallel architecture. Sequential, raster scanning algorithms can be implemented in parallel, offering an optimal computational cost (class D below). However, such an algorithm can not use more than a limited number of processors, so if a massively parallel architecture (i.e. one processor per pixel) is available, parallel algorithms in group B will produce a DT in shortest time.

C: Near optimal

2D: $O(N^2 \cdot \varepsilon(I))$
3D: $O(N^3 \cdot \varepsilon(I))$
In this group, we have all algorithms that in the worst case (certain pixel configurations in the image) run slower than optimal, but, at least for normal cases, faster than the parallel algorithms in class B. The computational cost is in $O(N^2 \cdot \varepsilon(I))$, where $\varepsilon(I)$ is a function that depends on the image data, implicitly dependent on and growing with N.

C1: Error free sequential EDTs

Most of the error free sequential EDTs described in Section 4 are in this class. The normally perform close to optimal, but have worst cases where sorting or queuing of many pixels give a higher computational complexity. Thus, $\varepsilon(I)$ can be strongly data dependent, approaching N for some kinds of images.

C2: Pyramid DTs

When computing DTs on parallel pyramid machines [65], an approximative DT can be computed in O(log(N)) iterations, which implies a computational cost in $O(N^2 \log(N))$, i.e. $\varepsilon(I) = \log(N)$.

As a side note, Borgefors et. al. [18] also use a pyramid structure, but choose to use it differently, computing the DT in parallel as usual (class B, in $O(N^3)$), using the pyramid for memory-efficient storage only.

C3: Sub-optimal ordered propagation DTs

Though ordered propagation DTs perform extremely well, some unexpected worst cases appear due to not processing the pixels in order of increasing distance. The early ordered propagation DTs by Piper and Granum [49] have such a worst case when using weighted metrics. The Constrained EDT, paper #3, also has such a worst case. See also class D2, below.

D: Optimal

2D: $O(N^2)$
3D: $O(N^3)$
The DTs in this class must have an upper limit on how many times any pixel can be accessed. If so, the time consumption will always be proportional to the image size (e.g. $O(N^2)$ for an $N \cdot N$ image). Two addressing mechanisms can be used, namely raster scanning or ordered propagation.

D1: Raster-scanning

The classical kind of raster scanning DTs, as proposed by Rosenfeld, are computationally optimal. It uses two image scans for all non-Euclidean DTs. Euclidean DTs use 3 or 4 scans. Rutowitz error-free EDT [63] is also in this class, using only 2 scans but with more arithmetic operations.

D2: Ordered propagation

Some, but not all ordered propagation DTs are computationally optimal. For an ordered propagation DT to be computationally optimal, it must access the pixels in order of increasing distance. If that is not fulfilled, we can find worst cases as discussed in paper #2, where some areas can be visited many times, where the upper bound depends on the image size.

The ordered propagation algorithm proposed by Verwer [68,69] is, however, optimal, since it accesses the pixels in perfect order of increasing distance. Paper #4 [54,54] discusses how to include this in an EDT algorithm. Another optimal algorithm, where optimality is less obvious, is the simpler ordered propagation EDT in paper #2 [52,53], which uses a distance threshold. It can not guarantee that every pixel is visited only once, but there is still an upper bound on the number of times a pixel can be accessed.

6. Going 3D

The task of making distance transformations in 3 or higher dimensions has been addressed in several papers. Mohr and Bajcsy [42] and Borgefors [16] proposed early generalizations of Danielsson's algorithm. Borgefors [16] also discussed other metrics extensively. Recently, Danielsson [25] proposed algorithms for octagonal distance in 3 dimensions.

In this thesis, I generally work in the 2-dimensional space, except for paper #7, which generalizes the results in paper #1 for use in 3 and arbitrary dimensions. The results in many of the other papers are more or less straight-forward to do in higher dimensions too. I will here briefly mention the situation for each paper, and then show how the less trivial extension of paper #2 can be done.

- Paper #1 is generalized to 3D in paper #7.
- For paper #2, some generalizations are given below.
- Extending paper #3 to higher dimensions is far from trivial, and I will make no attempt here to do that. It will not only require handling of direction intervals like in paper #7, but has to handle more complex direction intervals, i.e. polygons in the direction space.
- The shape editing operations in papers #4 and #5 are trivial to extend to 3D. As written, they may use the full 3·3 (in 3D 3·3·3) neighbourhood or directed masks, which I will describe how to do in 3D below.
- The local maxima extraction algorithm, paper #6, is somewhat harder to extend. In such a case, all the tables must be recalculated in order to give information about what *spheres* that overlap rather than disks.
- Paper #7 is in itself a discussion in 3D and higher dimensions.
- The skeletonizing algorithm in paper #8 is in itself easy to generalize to 3D, given a connectivity preserving test for 3-dimensional objects. An early method was proposed by Lobregt et. al. [38].
- The algorithm in paper #9 can be extended to 3D by replacing disks with spheres, but becomes even slower and thereby even less interesting for practical use. However, the more important question of the difference in generated data size between the simpler algorithms and an optimized one would be interesting to investigate in 3D.
- 3D is not relevant for paper #10, except possibly for the parallel algorithms.

The generalization of paper #2 is not quite as obvious as some of the others, and may deserve some comments. The paper describes an ordered propagation EDT with directed masks, that is, small, direction-dependent masks. In 2D, the direction space is divided in 8 parts. In 3D, it must be divided into more parts. Using the unfolded cube graph (introduced in paper #7), we can find a suitable division.

A mask with only 1 neighbour will only support propagation along a line. A mask with 2 neighbours can, as long as the two pixels are not located on a line from the center pixel, support propagation in a plane. A mask with 3 neighbours can support propagation

in a volume, a part of the 2-dimensional direction space that we have in 3D, granted that the three pixels and the center pixel are not all in the same plane. Thus, we need masks with at least 3 neighbours, and if possible we should not have more.

We get the simplest solution when using a 6-neighbour EDT. Then, we can use the same masks as in the Corner EDT (paper #7), but this time using an ordered propagation algorithm. See Figure 28. Each one of the eight masks is used in a direction interval like the shaded area in the Unfolded Cube Graph to the left in the Figure. The shaded area corresponds to the directions where the mask in the middle, the farthest bottom left one, should be used.

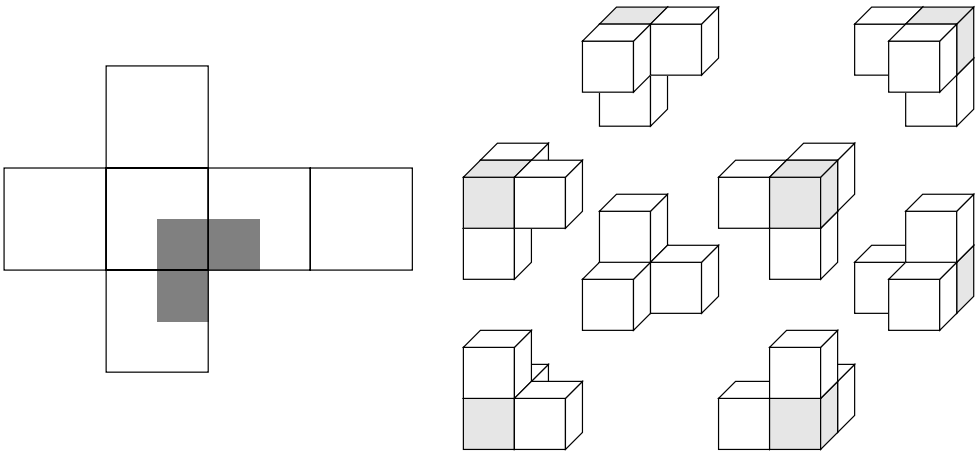


Figure 28. The eight masks in Corner EDT and the Unfolded Cube graph for one mask.

Just like in the 2D algorithm, we can apply one of these masks depending on what direction the current center pixel is in from its originating pixel. The directly supported directions are still a special case and requires special masks.

For no extra expense in computation speed, we can move up to the full 26-neighbourhood. See Figure 29. The Unfolded Cube graph in the figure is divided into triangles. Every corner of a triangle corresponds to one of the 26 neighbours. The mask corresponding to the direction space covered by the shaded triangle is shown to the right.

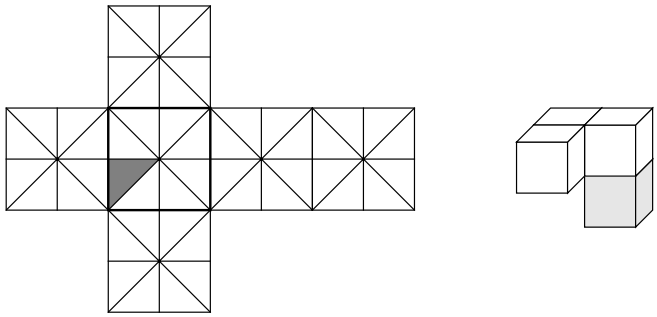


Figure 29. One possible division of the Unfolded Cube graph for an ordered propagation 3D EDT with 26-neighbourhood.

We can make the division of the direction space into triangles in different ways, i.e. splitting the squares the other direction, but the result will be the same. We may also divide it into squares, which gives us only half as many intervals, but that would imply four neighbours per mask rather than three.

Since we are still using only three neighbours per mask, and every pixel is processed only once due to the ordered propagation, this methods is just as fast as the 6-neighbour one. Of course, it takes a lot more work to implement.

We can conclude that it is possible to generalize the ordered propagation algorithm in paper #2 with directed masks to 3 and even higher dimensions, in 3D yielding algorithms visiting only 3 neighbours per pixel.

7. Applications

7.1 About this chapter

The distance transform is a tool that is applicable to a wide range of image processing problems. In this section, I will review a number of applications where the DT is a more or less central part of the solution. In Section 7.2, applications for the general distance transform, with only distance values, are reviewed. In Section 7.3, applications for labelled distance transform, producing distance maps with reference to the originating pixel or class of pixels, are reviewed.

7.2 Applications of distance maps in general

This section discusses applications of distance maps in general. Euclidean distance maps, both signed and unsigned, may be used as well as others. The applications does not explicitly demand labelled distance maps, but we will find a number of cases where such distance maps give better results.

Medial axis transform

The two following issues, data compression and skeletonization, are both based on the concept of the medial axis transform (MAT) also known as medial axis function (MAF) as suggested by Blum [11]. The MAT is defined for objects in the continuous space. The discrete skeleton (see below) is an approximation of the continuous MAT. Figure 30 shows the MAT of a few sample objects.

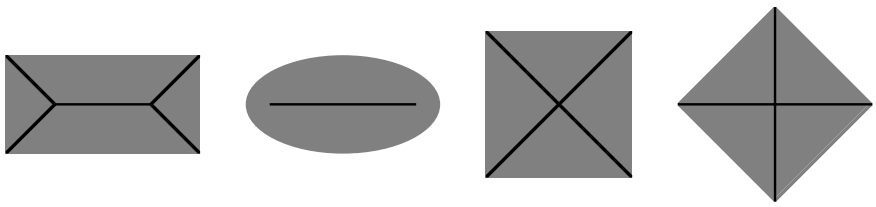


Figure 30. The Medial Axis transform of some sample objects.

Thus, MAT is in itself not a DT application, but a concept on which the following applications is based.

Data Compression

From a distance map over an object, it is possible to find a number of pixels who together with the corresponding distance values can uniquely represent the shape. Each such pixel corresponds to a (more or less) circular disk in the image. The shape can be reconstructed from it, according to the reconstruction theorem [61]. This is a compact representation of a shape.

In early papers, this shape representation was called *skeleton* or *distance skeleton*. Since the set of pixels needed for representing a shape is generally not connected, this was an unfortunate term, and the term skeleton is rather used for the connected skeleton today. Later, the term *local maxima* was used, referring to the fact that when using City

Block or Chessboard distance, all pixels needed for reconstruction are local maxima in the distance map. However, this was not a very good name either, since the necessary pixels are not local maxima in weighted DTs or Euclidean DTs. In paper #9 of this thesis, we propose the name *necessary maximal disks* (NMD).

The NMD set is a rather robust representation of a shape, except for very small shapes, where one pixel makes a big difference, as illustrated in Figure 31. Of course, the case where a single pixel is removed in the middle is a topology change and should make a big difference.

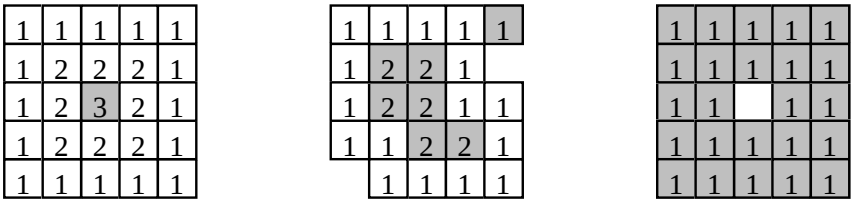


Figure 31. Some examples of maximal disk representations (skeletons) of some similar shapes (from Rosenfeld [61]) using the Chessboard distance metric.

Algorithms for generation of the NMD has been presented for City Block distance maps by Rosenfeld and Pfalz [62], for weighted metrics by Montanari [43] and Arcelli and Sanniti di Baja [4]. For Euclidean distance maps, algorithms have been proposed by Danielsson [22] and Borgefors et. al. [19,20] (paper #6). Mohr et. al. [42] use related ideas to represent 3-dimensional shapes with spheres, using a 3-dimensional version of the Euclidean distance transform. In paper #9 in this thesis we discuss the optimization of the NMD in order to get a minimal representation.

Skeletonization (thinning)

A topic related to the previous one is the connected skeleton. A connected skeleton is a topology-preserving representation of a shape in the form of one pixel wide curves along the medial axis of the shape. For some compact objects, the minimal representation as mentioned above is identical to the connected skeleton, but in the general case, neither connectedness nor single pixel width is guaranteed above, as exemplified by Figure 31. Connected skeletons are useful for shape analysis rather than data compression.

If connected skeletons are extracted by the NMD using metrics far from Euclidean, i.e. City Block or Chessboard distance, or an algorithm that operates by iterative topology preserving erosion [61], the skeletons generated are highly sensitive to both rotation of the represented object and to edge noise (See Figure 32 and Figure 33).

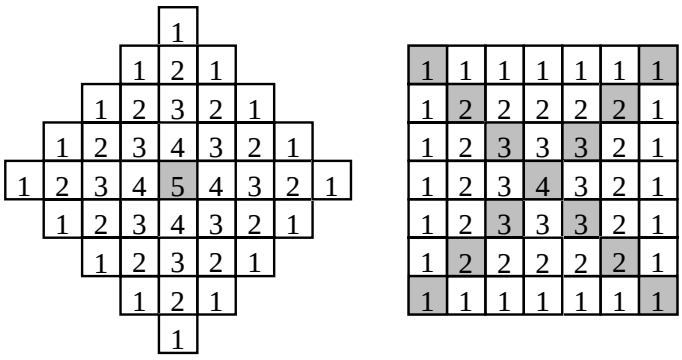


Figure 32. An example of the rotation dependency displayed by the skeletons generated by some thinning algorithms. The metric used is the City Block metric. The skeleton representation of the two squares are totally different.

Rotation invariance can be obtained by using the Euclidean metric or a close approximation of it. The sensitivity to edge noise can be suppressed by deleting short branches, but this is a heuristic solution that does not necessarily delete the right branches.

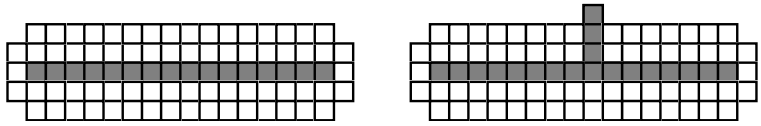


Figure 33. An example of the noise sensitivity of many thinning algorithms. A single pixel at the border adds a new branch in the skeleton, a considerable change of the representation of the shape.

We can reduce the rotation dependency by using better metrics, like the weighted metrics. Several such algorithms have been suggested [45,68]. For minimizing the rotation dependency, the Euclidean distance transform can be used. Fischler et. al. [30] and Arcelli et. al. [5] generate skeletons by filling in the NMD. In paper #8 [59], I propose an algorithm that accomplishes this task by connectivity preserving erosion on the EDM. Leymarie et. al. [36] use *snakes* along with Euclidean distance maps for extracting skeletons of similar high quality. The skeletons generated by this algorithm are both rotation invariant and noise insensitive.

There are also some algorithms that are rather approximations of a skeleton, where topology preserving is not guaranteed. Kruse [35] suggests the α-skeleton, where skeleton points must have neighbour pixels that hold vectors with an angle between them that is above a certain threshold α. Wright [73] uses a convolution kernel, the Marr Hidreth operator, to detect ridge points in a grey-level EDM.

Erosion or dilation in constant time

Erosion and dilation can be performed by iteratively changing border pixels of the objects in the processed image. However, since this is a parallel operation, it can be rather slow if we need to shrink or expand many steps.

If we generate a distance map over the suitable part of the image (objects or back-

ground), we may perform the erosion or dilation with a simple thresholding operation. The computation time will be far lower in many cases, since the distance map can be generated with a recursive (scanning or contour processed) DT. It will also allow us to use Euclidean or Quasi-Euclidean metrics.

Contour processed erosion and dilation

Another approach for fast erosion or dilation is contour processing binary operations (not DTs as above). The algorithms in [71] are fast even for erosion or dilation in many steps. However, unless we use very large structural elements, they bring us back to the crude City Block or Chessboard metrics again.

An alternative that gives high speed combined with Euclidean or Quasi-Euclidean metric is to use an ordered propagation (contour processed) DT that is halted after a specified distance. A Euclidean one is suggested in this thesis (Paper #2). With such an algorithm, we only need to process a small part of the image, corresponding to the area that should be influenced by the erosion or dilation.

PCB inspection

A practical application for erosion and dilation as well as skeletonization is automatic inspection of printed circuit boards (PCB). See for example [76]. However, the demand for speed in such applications is very large, so it is questionable whether distance transforms are fast enough compared to morphological operators implemented in hardware.

Shape factors

Danielsson [23] notes that shape factors based on the object border length are ill defined, because of the fractal behavior of object borders. It is not possible to make any unique measure of border length other than the convex hull. Danielsson suggests a new shape factor where the distance map of the object is used.

Edge matching

Distance maps can be used for efficient matching in binary images, as proposed by Barrow et. al. [7]. Borgefors [12,13] suggested an efficient algorithm using pyramid structures. Orbert [47] proposed some variations and reported that the choice of metric is of relatively small importance for this application.

The general technique is that we have a template, a shape defined as a set of pixels, and we want to find any positions in a binary image that matches the template. This can be done with correlation over the entire image (or GHT, generalized Hough transform, which is essentially the same thing), but this is a brute force solution, far too slow for real-time applications.

Instead, we make a distance map of the image. Then, we can try matching the template in some arbitrary position. The match is no longer just the sum of template pixels that end up on a feature pixel in the image, but rather an average of the distance values found. The smoothness of a distance map makes hill-climbing search for the optimal

match a reasonable approach that is likely to find the global optimum for a modest set of starting positions.

This method is very fast and fairly robust compared to correlation, though there are shapes where the global optimum is hard to find.

Path analysis

Distance maps can be used for path analysis, especially for finding the shortest path through an image with obstacles, using a constrained distance transform (Figure 34). Such algorithms are applicable to for example robot movement planning and PCB layout. The classical approach is described in the last part of this thesis. More recently, algorithms for path analysis with more accurate metrics have been presented [28,69]. However, the algorithms presented in the last part of this thesis are the first image-based algorithms known to the author that use the Euclidean metric, resulting in a path consisting of arbitrarily long straight lines.

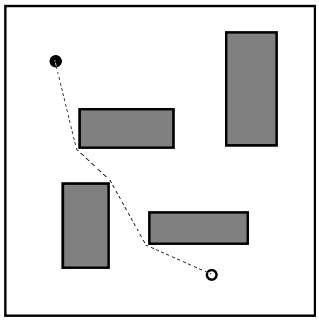


Figure 34. The result produced by a constrained distance transform, the shortest path through an image with obstacles (grey).

Skeletonization and dilation, which can be generated by distance transformations, are also useful for path planning. The skeleton of the background (non-obstacle areas) can be used for finding the safest path (the path that passes farthest from the obstacles). Dilation can be used for expanding the obstacles according to the size of the object that is to be moved through the area, reducing the problem to moving a point-shaped object through the area with the enlarged obstacles. If several different objects of different size should be used, we may keep the distance map instead of thresholding it, and use different thresholds depending on the object used. Andersen [6] suggests that the speed of the moving object could be dependent on such a distance map. When the object is close to obstacles, it should move slowly, in order to avoid collisions at high speed.

Textures, skeleton analysis

Toriwaki et. al. [67] use distance transformations of line patterns (DTLP) for texture analysis. This is a special kind of distance transformation, where each pixel in a line pattern receives the distance value to the closest or farthest end point (two different versions). See Figure 35. The transformation can be used for extracting information from connected skeletons.

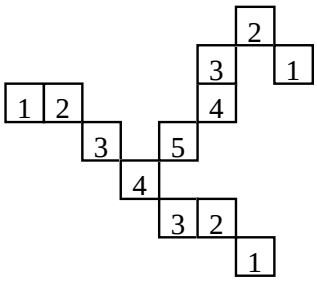


Figure 35. The distance transformation of line patterns.

It is possible to use weighted metrics like Chamfer 3-4 for the DTLP, but only as long as they are restricted to a 3·3 neighbourhood. More accurate distance measures like the Euclidean metric are not applicable on this kind of transformation without altering it completely.

The original DTLP was proposed as a parallel operation. It can also be computed sequentially [58], but in such a case, other methods appear more attractive.

Editing, smoothing

By erosion and dilation, edge noise can be removed from binary images. This can also be done using distance transformations, as suggested by Borgefors [15]. Arcelli and Sanniti di Baja [3] proposed a smoothing operation that uses a distance transformation of both object and background. In a related paper, Ablameyko et. al. [1] use distance maps for smoothing as well as topology changes. Ye [75] suggests a smoothing algorithm using the Euclidean metric.

In paper #5 in this thesis, I propose an improved version of the smoothing algorithm by Arcelli and Sanniti di Baja [3], using ordered propagation and Euclidean metric.

7.3 Applications of Euclidean distance maps and other labelled distance maps

The applications discussed in this section demand that the distance map holds information about where the distance values have propagated from. This is true for signed Euclidean distance maps, but also for other labelled distance maps. Note that non-Euclidean labelled distance maps have hardly any advantage over signed Euclidean distance maps, since the computation time and memory requirement are similar, but Euclidean distance maps have far better precision.

Voronoi diagrams

The Voronoi Diagram [31] is easily extracted from a labelled distance map. See Figure 25 for an example. Using distance transformations, Voronoi diagrams in the form of sampled images can be generated in constant time, regardless of the number of object points. This is done for non-Euclidean metrics by Borgefors [14] and for the Euclidean metric by Ye [75]. A large number of applications of Voronoi diagrams exist.

Shape analysis

Distance transformations can be used for other kinds of shape representation and shape analysis than the skeleton. Ito et. al. [32] present an algorithm to decompose shapes into significant, possibly overlapping substructures. Similar ideas are suggested by Mauer et. al. [40].

Sketch completion

In image analysis concerned with real images, parts of objects can be undetectable. Objects may overlap each other and edges can be too blurred to detect. Fischler et. al. [30] suggest that we may use sketch completion to reconstruct the shapes. They suggest a method using labelled distance maps. If only parts of the edges of an object is detectable, the outer and inner sides of the edges are labelled with two different labels. A distance map from these two regions and the following segmentation of the distance map according to the labels give a completed object. See Figure 36.

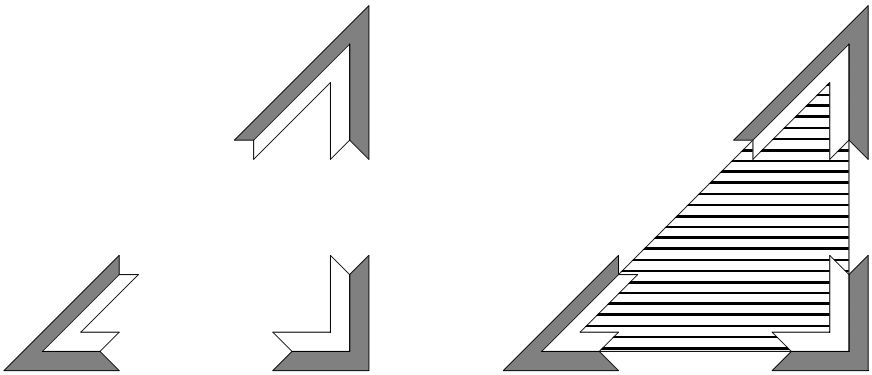


Figure 36. Sketch completion.

The left part of the figure shows the edges, the outer parts of which are labelled with one label (grey) and the inner parts with another label (white). Note that the edge segments are “notched” in the ends. The propagation from the white labelled parts of the edges will cover the striped area shown in the right figure.

In order to get good results, high precision metrics like the Euclidean metric is desirable, but it is also important to preprocess the ends of the edge segments, or artifacts will appear.

Curvature measurement

Ye [75] uses signed Euclidean distance maps for measuring curvature. From the distance map, we can find how many pixels in a certain distance interval that points to each object pixel. See Figure 37.

?	6	?	7	7	7	7	7	10	11	11	11	14	14	14	14	14
5	?	6	?	7	7	7	9	10	11	11	11	14	14	14	14	14
5	5	?	6	?	7	?	9	10	11	11	13	14	14	14	14	14
5	5	5	?	6		8	9	10	11	?	13	14	14	14	14	14
5	5	5	5							12	13	14	14	14	14	14
4	4	4	4									15	15	15	17	17
3	3	3	3									16	?	17	17	17
2	2	2	2										17	17	17	17
1	1	1	1										18	18	18	18

Figure 37. Labelled distance maps used for curvature measurement.

The figure shows a part of an object (grey). Each pixel in the area closest to the object has been labelled, showing for each edge pixel the background pixels for which that edge pixel is the closest one. Basically, this is a sampled Voronoi diagram. The larger the number of background pixels in a region, the higher the curvature.

This is done by scanning the image while accumulating values in a curvature image. This operation can be speeded up significantly by using the contour processed Euclidean distance transform, executing only a few iterations. Fischler et. al. [30] suggests a similar method, where the count of each pixel is weighted by the inverse of the distance.

Restitution

Blom [10] uses labelled distance maps for restitution, where the image of a face is distorted to simulate the result of facial surgery.

The input to the restitution algorithm is the face contours before and after surgery and the complete greyscale image of the face before surgery. The result is a greyscale image predicting the result of the surgery. Each contour pixel of any of the two face contours is given a label, that is, the contour pixels are used as object pixels in a distance transformation. Through a labelled distance map, the closest contour pixel is found for each pixel in the face area.

The restitution is done depending on the distance and direction to the contour for each pixel and the distance between two corresponding pixel in each contour. The farther the contours are, and the closer a pixel is to the contour, the larger displacement should be used. Hence, the displacement is depending on distance to the contour, the distance between the corresponding contour pixels and the direction of the contour displacement.

The output image is generated by, for each pixel in the output image, finding a displacement from the distance map and the contour data, which gives a pointer to the pixel whose value should be used.

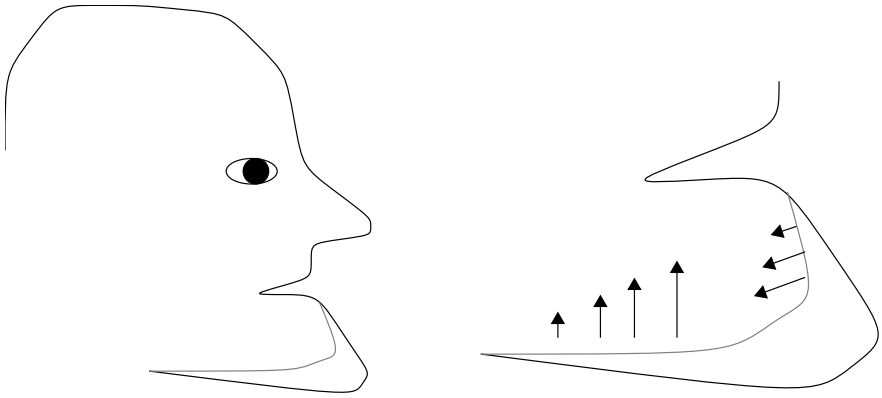


Figure 38. Surgery planning by restitution using labelled DT.

Figure 38 illustrates the process. The chin is to be shortened to the shaded curve. Blom's restitution algorithm introduces displacement of the pixels in different parts of the face image depending on the distance to the face contour and the relative position of corresponding contour pixels. Some hypothetical displacements are shown in the enlarged figure, right.

Blom notes that the number of edge pixels are limited by the word length of the labels, typically 8 bits, implying a maximum of 256 edge pixels. Using an Euclidean distance map, however, the memory requirement would be the same but there would not be any practical limit for the number of edge (object) pixels. The same holds for many other applications of labelled distance maps.

7.4 Summary of application areas

Distance maps in general and Euclidean distance maps in particular are extremely useful tools in image analysis and can be used for solving a broad variety of problems, including but not limited to the following list:

- Data compression
- Skeletonization
- Fast Erosion or dilation
- Form factors
- Chamfer matching
- Path analysis
- Textures
- Voronoi diagrams
- Region organization
- Sketch completion
- Curvature measurement
- Restitution

8. What is in this thesis?

Disregarding this introductory part, this thesis consists of ten technical papers, three of which were also in my licentiate thesis. They are listed in chronological order according to the first complete version of each paper. Some papers have been extensively revised since the first version, often shortened.

Paper #1

The Euclidean Distance Transformation and its implementation on SIMD architectures

A previous, longer version was included in my licentiate thesis [50].
A short version was presented at the 6th Scandinavian Conf. on Image Analysis [51].
This paper explains why the original EDT needs 4 scans while pseudo-Euclidean DTs generally only need two. I use the results to make a 3-scan EDT algorithm and EDT algorithms that are useable on 1-dimensional SIMD architectures.

Paper #2

Neighbourhoods for Distance Transformations using Ordered Propagation

A previous, longer version was included in my licentiate thesis [50].
A short version was presented at the 5th Int. Conf. on Image Analysis and Processing [52].
The present version has been published in CVGIP: Image Understanding [53].
In this paper, I investigate methods for optimizing the computation of EDT and pseudo-Euclidean DT on sequential computers, using ordered propagation while accessing only 1-3 neighbours for each pixel. In theory, this would give us a 3-4 times speed-up, though the complexity of the algorithm reduces the speed-up to about 2, which still is a considerable improvement.

Paper #3

The Constrained Euclidean Distance Transform

This paper was included in my licentiate thesis [50] in a form very close to the present paper.
A short version was presented at the SSAB symposium 1990.
In constrained DTs, obstacles are allowed. Adapting the EDT for this problem is a rather hard task, which is addressed in this paper. The solution is not exactly simple, but the algorithm produces an output that is much more compact than conventional grid-based methods.

Paper #4

Fast erosion and dilation by contour processing and thresholding of distance maps

Presented at the 7th Scandinavian. Conf. on Image Analysis [54].
Published in Pattern Recognition Letters [54].
Here, the EDT is applied to the erosion and dilation problem. Rather than just running DTs over a binary image and then applying a threshold, I have made an integrated algorithm that doesn't even use a real distance map, but holds the information implicitly stored in the propagation front.

Paper #5

Fast edge smoothing in binary images using Euclidean metric

Presented at the 6th Int. Conf. on Image Analysis and Processing [56].
This paper is closely related to the previous paper, but adapts the techniques used there for improving an edge smoothing algorithm (one that is not based on opening and closing).

Paper #6

The Euclidean Distance Transform: Finding the Local Maxima and Reconstructing the Shape

Written in cooperation with Gunilla Borgefors and Gabriella Sanniti di Baja.
Presented at the 7th Scandinavian Conf. on Image Analysis [19].
We present methods for finding local maxima (named Necessary Maximal Disks in paper #9), that is, the minimal set of disks from which the shape can be reconstructed, by using EDT, and also propose a fast reconstruction method.

Paper #7

The Euclidean Distance Transform in arbitrary dimensions

Presented at the 4th Int. Conf. on Image Processing and its Applications [57].
Accepted for publication in Pattern Recognition Letters.
The results in paper #1 are here generalized to 3 and higher dimensions. The solution in 3 dimensions turns out to be a 4-scan algorithm, which is found to be the minimal number of scans for 3D.

Paper #8

Rotation invariant skeletonization by thinning using anchor points

Accepted for the 8th Scandinavian Conf. on Image Analysis [59].
The possibility of using the set of local maxima (see also papers #6 and #9) and Kruse's α-skeleton [35] for topology preserving skeletonizing is investigated. The result is a grid-based skeleton, connectivity preserving and rotation invariant within the limits imposed by the discrete grid. The algorithm always runs in a time proportional to the number of pixels in the image. An integrated 1-pass algorithm is outlined.

Paper #9

Towards a minimal shape representation using maximal disks

Written in cooperation with Gunilla Borgefors.
Not yet published.
The number of Necessary Maximal Disks (local maxima) extracted by methods proposed so far is much larger than the true minimum. By using an optimizing algorithm, we show that the minimal number of necessary maximal disks is generally substantially lower than what is usually extracted.

Paper #10

A note on "Optimization on Euclidean Distance Transformation Using Grayscale Morphology"

Submitted to Journal of Visual Communication and Image Representation.
This paper describes errors in a supposedly error-free Euclidean DT that was recently published. The algorithm turns out to be subject to errors from a number of sources, which makes the analysis of these errors an exercise with tutorial value.

References

[1] S. Ablameyko, C. Arcelli, G. Sanniti di Baja, “Using distance information for editing binary pictures”, *Proc. 6th Scandinavian Conference on Image Analysis*, Oulo, 1989, pp 401-407.

[2] S.G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Eaglewood Cliffs, New Jersey, 1989.

[3] C. Arcelli, G. Sanniti di Baja, “Picture editing by simultaneously smoothing figure protrusions and dents”, *Proceedings of the 9th International Conference on Pattern Recognition*, Rome, 1988, pp 948-950.

[4] C. Arcelli, G. Sanniti di Baja, “Finding local maxima in a pseudo-Euclidean distance transform”, *Computer Vision, Graphics and Image Processing* 43, 1988, pp 361-367.

[5] C. Arcelli, G. Sanniti di Baja, “Ridge points in Euclidean distance maps”, *Pattern Recognition Letters 13*, 1992, pp 237-243.

[6] J.D. Andersen, “Robot control based on neighborhood image operations”, *IFAC Digital Image Processing in Industrial Applications*, Espoo, Finland, 1986.

[7] H.G. Barrow, J.M. Tenenbaum, R.C. Bolles, H.C. Wolf, “Parametric correspondence and Chamfer Matching: Two new techniques for image matching”, *Proc. 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, USA, 1977, pp 659-663.

[8] A.L.D. Beckers, A.W.M. Smeulders, “Optimization of length measurements for isotropic distance transformations in three dimension” (sic.), *CVGIP:Image Understanding* 55, 1992, pp 296-306.

[9] A.L.D. Beckers, A.W.M. Smeulders, “A comment on ‘A note on “Distance transformations in digital images”’”, *Computer Vision, Graphics and Image Processing* 47, 1989, pp 89-91.

[10] A. Blom, *Planering och simulering av käkkirurgi med bildbehandlingsdator*, Examensarbete (Masters Thesis), LiTH-ISY-EX0865, Linköping University, 1988 (in swedish).

[11] H. Blum, “A transformation for extracting new descriptors of shape”, *Models for the Perception of Speech and Visual Form*, MIT Press, Cambridge, Mass., 1967, pp 362-380.

[12] G. Borgefors, “An improved version of the Chamfer Matching algorithm”, *Proc. 7th International Conference on Pattern Recognition*, Montreal, 1984, pp 1175-1177.

[13] G. Borgefors, *On Hierarchical Edge Matching in digital images using Distance Transformations*, Dissertation, FOA Report B30094-E1, 1986.

[14] G. Borgefors, “Distance transformations in digital images”, *Computer Vision, Graphics and Image Processing* 34, pp 344-371, 1986.

[15] G. Borgefors, “Distance transformations in digital images”, FOA Report, C 30401-E1, 1985.

[16] G. Borgefors, “Distance transformations in arbitrary dimensions”, *Computer Vision, Graphics and Image Processing* 27, pp 321-345, 1984.

[17] G. Borgefors, “Distance transformations in hexagonally digitized images”, FOA Report C30497-3.3, 1988.

[18] G. Borgefors, T. Hartmann, S.L. Tanimoto, “Parallel pyramidal distance transforms on pyramid machines: theory and implementation”, *Signal Processing* 21, 1990, pp 61-86.

[19] G. Borgefors, I. Ragnemalm, G. Sanniti diBaja, "The Euclidean Distance Transform: Finding the local maxima and reconstructing the shape", *Proceedings: The 7th Scandinavian Conference on Image Analysis*, Aalborg, 1991, pp 974-981.

[20] G. Borgefors, I. Ragnemalm, G. Sanniti diBaja, "Feature extraction on the Euclidean Distance Transform", in: Cantoni et. al, eds, *Progress in Image Analysis and Processing II*, World Scientific, Singapore, 1992, pp 115-122.

[21] G. Borgefors, “Another comment on ‘A note on “Distance transformations in digital images”’”, *CVGIP: Image Understanding* 54, 1991, pp 301-306.

[22] P.E. Danielsson, “Euclidean Distance Mapping”, *Computer Graphics and Image Processing* 14, 1980, pp 227-248.

[23] P.E. Danielsson, “A new shape factor”, *Computer Graphics and Image Processing* 7, 1978, pp 292-299.

[24] P.E. Danielsson, P. Lord, “Application of distance maps on binary images”, *Proc. 1st Scandinavian Conference on Image Analysis*, 1980, pp 50-56.

[25] P.E. Danielsson, “Minimal error octagonal metric in two and three dimensions”, accepted for 8th Scandinavian Conf. on Image Analysis, 1993.

[26] P.P. Das, “Octagonal distances for digital pictures”, *Information Sciences* 50, 1990, pp 123-150.

[27] *Dictionary of Scientific and Technical Terms*, 4th edition, McGraw-Hill, 1989.

[28] L. Dorst and P.W. Verbeek, “The constrained distance transformation: A pseudo-Euclidean, recursive implementation of the Lee-algorithm”, *Signal Processing III: Theories and Applications* (Proc. European Signal Processing Conf.), I.T. Young et. al., eds, Elsevier Science Publ. B.V., 1986, pp 917-920.

[29] *Encyclopædia Britannica*, 14th edition, 1929, pp 802-803.

[30] M.A. Fischler, P. Barrett, “An iconic transform for sketch completion and shape abstraction”, *Computer Graphics and Image Processing* 13, 1980, pp 334-360.

[31] L. Guibas and J. Stolfi, “Primitives for the manipulation of general subdivisions and the computation of Voronoi Diagrams”, *ACM Transactions on Graphics*, Vol. 4, No 2, April 1985, pp 74-123.

[32] Y. Ito, K. Abe, C. Arcelli, “Region organization in two-dimensional shapes”, in: Cantoni et. al., *Progress in Image Analysis and Processing, World Scientific, Singapore, 1990*, pp 332-339.

[33] F. Klein, O. Kübler, “Euclidean distance transformations and model-guided image interpretation”, *Pattern Recognition Letters* 5 (1987), pp 19-29.

[34] F. Klein, “Euclidean Skeletons”, *Proceedings of the 5th Scandinavian Conference on Image Analysis*, Stockholm, 1987, pp 443-450.

[35] B. Kruse, “An exact sequential Euclidean distance algorithm with application to skeletonizing”, Proceedings, *7th Scandinavian Conf. on Image Analysis*, Aalborg, 1991, pp 982-992 (revised version of an Internal Report, Teragon Systems, 1987).

[36] F. Leymarie, M.D. Levine, “Simulating the Grassfire Transform using an active contour model”, *IEEE Trans. on Pattern Analysis and Machine Intelligence 14*, pp 56-75.

[37] F. Leymarie, M.D. Levine, “Fast raster scan distance propagation on the discrete rectangular lattice”, *CVGIP:Image Understanding* 55, 1992, pp 84-94.

[38] S. Lobregt, P.W. Verbeek, F.C.A. Groen, “Three-dimensional skeletonization: Principle and algorithm”, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol 2, no 1, 1980, pp 75-77.

[39] E. Luczak, A. Rosenfeld, “Distance on a hexagonal grid”, *IEEE Trans. on Computers*, may 1976, pp 532-533.

[40] E. Mauer, R. Schaerf, “New applications of distance transformation methods for effective structural image analysis”, *Proceedings of the 8th International Conference on Pattern Recognition*, 1986, pp 666-668.

[41] *McGraw-Hill Encyclopedia of Science and Technology*, 7th edition, 1992.

[42] R. Mohr, R. Bajcsy, “Packing Volumes by Spheres”, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol 5, no 1, 1983, pp 111-116.

[43] U. Montanari, “A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance”, *Journal of the ACM, vol 15*, 1968, pp 600-624.

[44] J. Mullikin, “The Vector Distance Transform in two and three dimensions”, *CVGIP: Graphical Models and Image Processing* 54, 1992, pp 526-535.

[45] C.W. Niblack, P.B. Gibbons, D.W. Capson, “Generating skeletons and centerlines from the distance transform”, *CVGIP: Graphical Models and Image Processing* 54, No 5, 1992, pp 420-437.

[46] R. Ogniewicz, M. Ilg, “Skeletons with Euclidean metric and correct topology and their application in object recognition and document analysis”, *Proc. 4th Int. Symposium on Spatial Data Handling*, Zürich, 1990, pp 15-24.

[47] C.L. Orbert, “Iterative methods for edge matching using distance transformations”, *ICARCV-92*, 1992, pp CV-3.3.1-5.

[48] D. Paglieroni, “A unified distance transform algorithm and architecture”, *Machine Vision and Applications 5*, 1992, pp 47-55.

[49] J. Piper, E. Granum, “Computing distance transformations in convex and non-convex domains”, *Pattern Recognition* 20, *1987*, pp 599-615.

[50] I. Ragnemalm, *Generation of Euclidean Distance Maps*, Linköping Studies in Science and Technology, Thesis No. 206 (licentiate thesis), Linköping, Sweden, January 1990.

[51] I. Ragnemalm, "The Euclidean Distance Transform and its Implementation on SIMD architectures", *Proc. 6th Scandinavian Conference on Image Analysis*, Oulo, 1989, pp 379-384.

[52] I. Ragnemalm, “Contour processing distance transforms”, in: Cantoni et. al., *Progress in Image Analysis and Processing, World Scientific, Singapore, 1990,* pp 204-212.

[53] I. Ragnemalm, “Neighborhoods for distance transformation using ordered propagation”, *CVGIP: Image Understanding* 56, no 3, 1992, pp 399-409.

[54] I. Ragnemalm, “Fast erosion and dilation by contour processing and thresholding of distance maps”, *Proc. 7th Scandinavian Conference on Image Analysis*, Aalborg, 1991, pp 173-180.

[55] I. Ragnemalm, “Fast erosion and dilation by contour processing and thresholding of distance maps”, *Pattern Recognition Letters* 13, 1992, pp 161-166.

[56] I. Ragnemalm, “Fast edge smoothing in binary images”, in: Cantoni et. al, eds, *Progress in Image Analysis and Processing II, World Scientific, Singapore*, 1992, pp 83-90.

[57] I. Ragnemalm, “The Euclidean Distance Transform in arbitrary dimensions”, *Proceedings, 4th Internal Conference on Image Processing and its Applications*, Maastricht, 1992, pp 290-293, accepted for publication in Pattern Recognition Letters.

[58] I. Ragnemalm, S. Ablameyko, “On the distance transformation of line patterns”, accepted for 8th Scandinavian Conf. on Image Analysis, 1993.

[59] I. Ragnemalm, “Rotation invariant skeletonization by thinning using anchor points”, accepted for 8th Scandinavian Conf. on Image Analysis, 1993.

[60] F. Rhodes, “Discrete Euclidean metrics”, *Pattern Recognition Letters* 13, 1992, pp 623-628.

[61] A. Rosenfeld, A.C. Kak, *Digital Picture Processing*, Academic Press, New York, 2nd edition, 1982

[62] A. Rosenfeld, J.L. Pfaltz, “Sequential operations in digital picture processing”, *Journal of the ACM* 13, No 4 1966, pp 471-494.

[63] D. Rutowitz, “Efficient processing of 2-D images”, in: Cantoni et. al, eds, *Progress in Image Analysis and Processing II, World Scientific, Singapore*, 1992, pp 229-253.

[64] D. Rutowitz, personal communication.

[65] S. Tanimoto, P.E. Danielsson, “On the convergence of a pyramidal Euclidean distance transform”, *Proc. Int. Workshop on Comp. Arch. for Machine Perception* 1991 (CAMP’91), pp 285-296.

[66] J.I. Toriwaki, S. Yokoi, “Distance transformations and skeletons of digitized pictures with applications”, *Progress in Pattern Recognition*, L.N. Kanal and A. Rosenfeld (editors), North-Holland Publishing Company, 1981, pp 187-264.

[67] J.I. Toriwaki, N. Kato, T. Fukumura, “Parallel local operations for a new distance transformation of a line pattern and their applications”, *Proceedings of the 4th International Conference on Pattern Recognition*, Kyoto, 1978, pp 649-653.

[68] B.J.H. Verwer, “Improved metrics in image processing applied to the Hilditch Skeleton”, *Proceedings, 9:th International Conf. on Pattern Recognition*, 1988, pp 137-142.

[69] B.J.H. Verwer, P.W. Verbeek and S.T. Dekker, “An efficient uniform cost algorithm applied to distance transforms”, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol II, No 4, April 1989, pp 425-429.

[70] L. Vincent, “Exact Euclidean distance function by chain propagation”, *Proc. Computer Vision and Pattern Recognition ’91*, 1991, pp 520-525.

[71] L.J. van Vliet and B.J.H Verwer, “A contour processing method for fast binary neighbourhood operations”, *Pattern Recognition Letters* 7, 1988, pp 27-36.

[72] A.M. Vossepoel, “A note on ‘Distance transformations in digital images’”, *Computer Vision, Graphics and Image Processing* 43, 1988, pp 88-97.

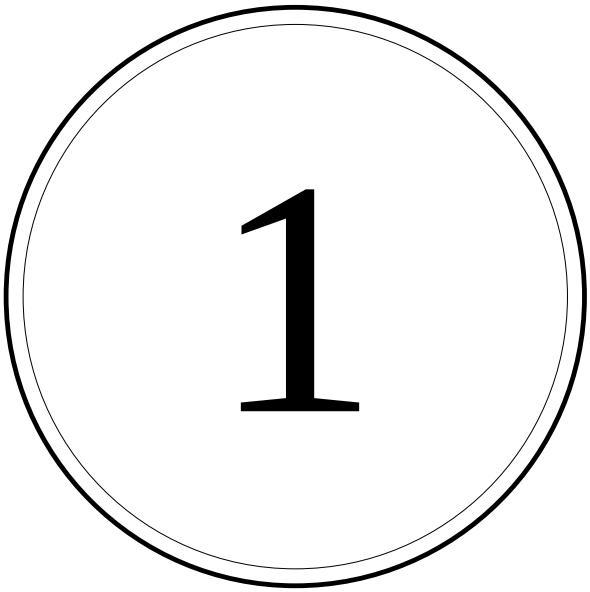
[73] M.W. Wright, “Skeletonization as model based feature detection”, *Proceedings, 4th Internal Conference on Image Processing and its Applications*, Maastricht, 1992, pp 254-257.

[74] H. Yamada, “Complete Euclidean Distance Transformation by parallel operation”,

Proceedings, 7:th International Conference on Pattern Recognition, 1984, pp 69-71.

[75] Q.Z. Ye, “The Signed Euclidean Distance Transform and its applications”, Proceedings, *9:th International Conference on Pattern Recognition,* 1988, pp 495-499.

[76] Q.Z. Ye, P.E. Danielsson, “PC-board checking algorithms using Connectivity Preserving Shrinking“, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Sept. 1988.



The Euclidean Distance Transform and its implementation on SIMD architectures

Short version in: Proceedings, 6th Scandinavian Conference on Image Analysis, Oulo, 1989.

The Euclidean Distance Transform and its implementation on SIMD architectures

Ingemar Ragnemalm

Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden.E-mail: ingemar@isy.liu.se

ABSTRACT:

The recursive versions of the Signed Euclidean Distance Transform can not be implemented on parallel architectures with any speed improvement. This paper describes new versions of the algorithm that can be implemented on a 1-dimensional SIMD architecture, taking advantage of the parallelism. The algorithms promise to give a much better price to performance ratio compared to parallel distance transformation algorithms. The paper also contains a discussion about the propagation of distance values in distance transforms, especially Euclidean ones, stating a sufficient condition for Euclidean distance transforms to work properly.

1. Introduction

1.1 Parallel and sequential algorithms.

Distance transforms are useful tools in image processing concerned with binary images. The operation creates a distance map which for each reset pixel holds a distance value to the closest set pixel (henceforth referred to as *object pixel*).

The simplest way to compute distance transforms is by using global operations, where for every pixel, the distances to all object pixels are calculated in order to find the closest one. Such an operation would take a time $\in O(N^4)$ for a N^2 image with $O(N^2)$ object pixels. However, in order to reduce computational complexity the distance maps are usually created by iterative algorithms where the distance values propagate in small steps using neighborhood operations. Such algorithms may be either *parallel* or *sequential*.

In parallel algorithms, neighborhood operations are applied simultaneously on all pixels, while in sequential algorithms only *one* operation on one pixel and its neighborhood at a time is executed. A parallel algorithm may be implemented sequentially, using only one processor. Conversely, but less obviously, some sequential algorithms may be implemented on parallel architectures, employing several processors. This is the subject of the current paper.

In parallel algorithms, the distance value of each pixel is compared to values from neighbors in all directions, and the lowest value is written to the pixel. Hence the operation gives propagation in all directions, as illustrated by Figure 1. The operation is repeated until no changes has occurred during an iteration. Assuming the maximum distance to grow with image size, this takes a number of iterations $\in O(N)$, which implies a computational complexity of $O(N^3)$.

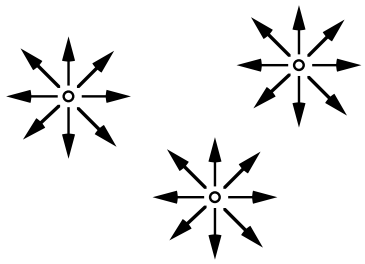


Figure 1. In parallel algorithms, the distance values propagate in all directions simultaneously.

Sequential algorithms scan the picture pixel by pixel and makes comparisons with some neighbors. After a limited number of scans (typically 2), the distance map is complete. Hence, the computation time is $\in O(N^2)$. The scanning procedure is illustrated in Figure 2. In Figure 2a, the forward scan is illustrated, scanning each row from right to left, starting with the uppermost row. This will give all pixels within a 90˚ arc from the object pixel their correct distance values (right part of the Figure). The backwards scan, Figure 2b, will propagate to other areas.

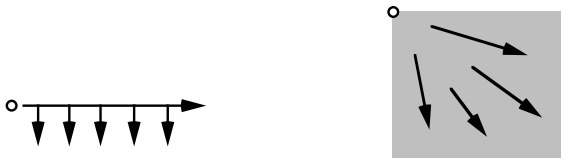


Figure 2a. First scan of a raster scanning DT.

Figure 2b. Second scan of a raster scanning DT.

In both parallel and sequential algorithms the actual neighborhood operation is performed as follows. The values to which we compare the distance value of the center pixel are the distance values of the neighbors, modified to get the distance from the center pixel to the object pixel instead of to the neighbors. In its simplest form, the neighbor value is increased by one. The comparison may then be written as (1), where the distance value of pixel 1 is set to the minimum of a) its own present value and b) the value of the neighbor, pixel 2, increased by one. Throughout this text, propagation of distances is always supposed to include this kind of elementary operation in some form or another.

$$D_{pixel1} = \text{Min}\,(D_{pixel1}\,,\, D_{pixel2} + 1) \quad (1)$$

Figure 2 suggests that the algorithm makes comparisons with the neighbors one step down and one step to the right in the first scan, and up and left in the second scan. We may represent the neighborhoods involved with the two *masks* shown in Figure 3. The 1's correspond directly to the +1 increment in (1).

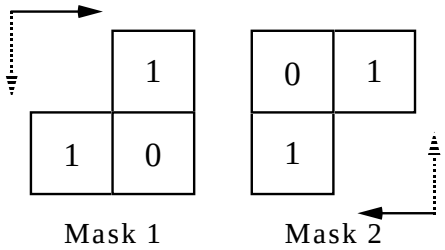


Figure 3. Masks for a simple distance transform algorithm, using City Block distance. 0 is the center pixel, the one we seek a better (lower) value for. Dotted arrows show the direction of the outer loop, that is, we will scan across the picture once in the direction of the solid arrows for each step along a dotted arrow.

The algorithm of Figure 3 uses the City Block distance metric. In pseudo code, this can be expressed as shown below.

Example of a simple sequential algorithm (City Block distance)

```
[First picture scan]
for row=1..R-1
for column=1..K-1
                          dist(row, column)
dist(row, column)=min     dist(row-1,column)+1
                          dist(row,column-1)+1

[Second picture scan]
for row=R..2
for column=K..2
                          dist(row, column)
dist(row, column)=min     dist(row+1,column)+1
                          dist(row,column+1)+1
```

For a small picture with only one object pixel, the two picture scans will give the results shown in Figure 4.

```
- - - - -        4 3 2 3 4
- - - - -        3 2 1 2 3
- - 0 1 2        2 1 0 1 2
- - 1 2 3        3 2 1 2 3
- - 2 3 4        4 3 2 3 4
```

Figure 4. The distance map after the first scan will cover only a part of the picture. The second one will fill out the rest of the picture with correct City Block distance values.

This algorithm is essentially the same as the sequential distance transformation algorithm, presented by Rosenfeld [9] in 1966.

Let us make a short parenthesis: when we process a pixel, we have got the choice between “propagating out from the pixel” and “propagating into the pixel”. In the first case, we compute distance values for each member of the mask from the value of the center pixel, and update any of them *except* the center pixel. In the second case a distance value for the center pixel is computed from the distance value of each member, and update *only* the center pixel. The second method has some advantage in causing less unnecessary updates, and is probably the most commonly used.

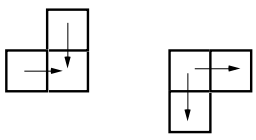


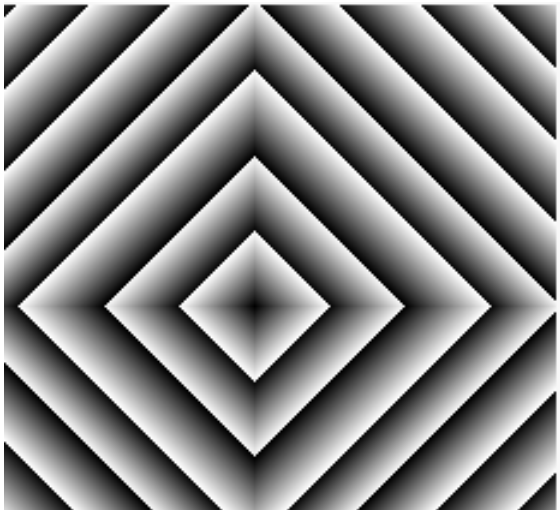
Figure 5. Distance values may either propagate from the neighbors to the center pixel or from the center pixel to its neighbors (The figure illustrates a sequential case.)

Verwer names these two methods “write formalism” and “read formalism”, respectively [12].

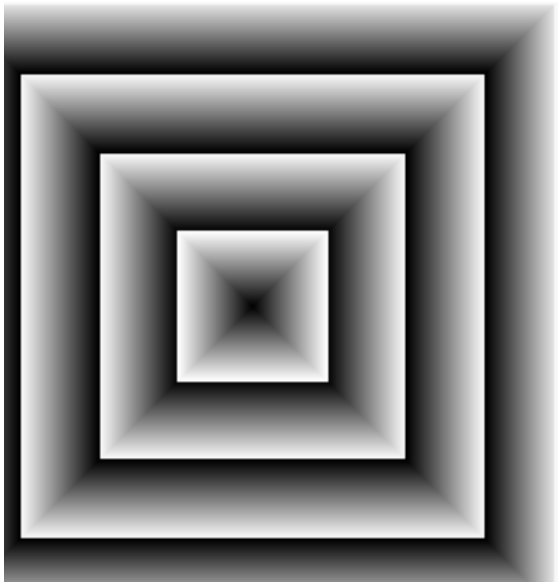
1.2 Euclidean distance transforms

The simplest of all distance transforms are the transforms based on the City Block distance metric. One such algorithm was described in the previous section. A number of different metrics are illustrated in Figure 5 a-e, showing distance maps from a single object pixel. The greyscale shows the distance value in a modulo fashion (a number of the highest bits set to zero) in order to make the accuracy more visible. The stripes in the figures (the equidistance curves) should ideally be absolutely circular.

Figure 5. Distance transforms from a single object pixel with different metrics.



5a. City Block distance transform.

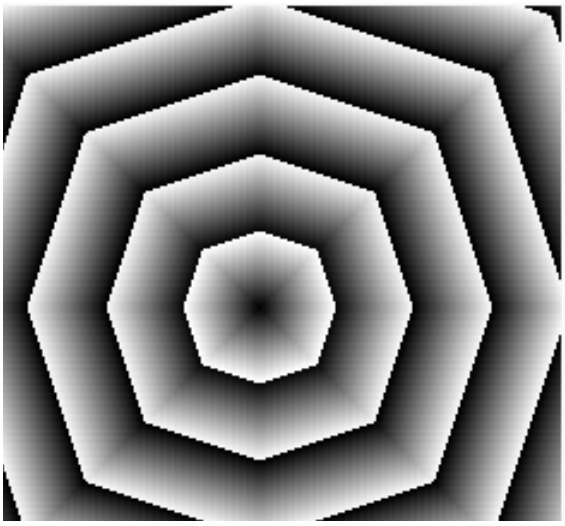


5b. Chessboard distance transform.

It is obvious from Figure 5a that City Block distance as well as Chessboard distance are far from the Euclidean distance. The equidistance curves are quadratic. The Chessboard distance, illustrated in Figure 5b, has errors of the same magnitude, but demands somewhat more computation time.

For a long time, only these approximative distance maps could be created by the fast, propagating algorithms. Methods for creating close approximations to Euclidean distance were developed [2, 7], but for all of them the errors increased with increasing distances.

The Chamfer distance algorithms [2], using different distance values for different pixels in the masks, make far better approximations of the Euclidean distance, and are often referred to as Quasi-Euclidean distance. The simplest one, Chamfer 3-4, demands about the same computation time as the Chessboard distance. It is illustrated in Figure 5c. Note that the equidistance curves are octagons.



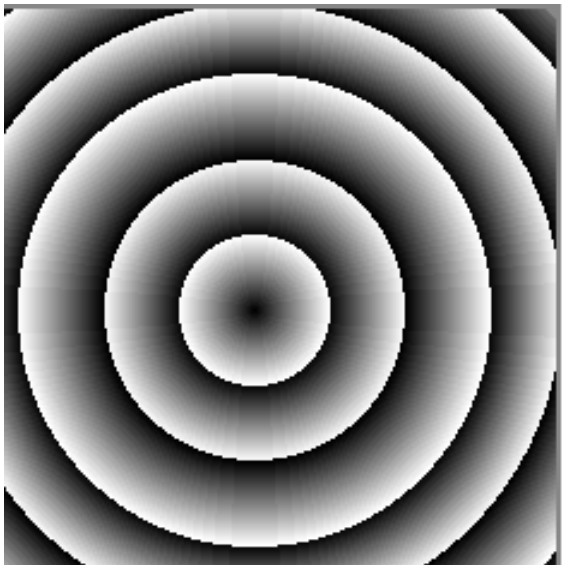
5c. Chamfer 3-4 distance transform.

Even better approximations are created by using larger neighborhoods. The Chamfer 5-7-11 distance, illustrated in Figure 5d, uses masks of 5·5 pixels. In this case, the equidistance curves are polygons with 16 corners.

The Euclidean Distance Transform (EDT) was first proposed by Danielsson [3] in 1980. Danielsson's sequential algorithm proves not to be totally error-free, but the errors are small (far less than a pixel distance), bounded (actually decreasing with distance) and occurs only in very few discrete pixels. This is discussed further in section 2.5.

5d. Chamfer 5-7-11 distance transform.

Figure 5e illustrates the Euclidean distance map from a single object pixel, computed with the original EDT algorithm. There are no errors whatsoever. The distance map is exact. The equidistance curves are perfect circles (though in a sampled grid, of course).



5e. Euclidean distance transform.

Both Danielsson [3] and Yamada [10] point out the possibility to include directions to object pixels in the Euclidean distance map. This variant is named *Signed Euclidean Distance Transform* (SEDT). A number of new possibilities arise from having this information available. This was investigated by Ye [11], who developed a number of applications, for example a convex hull algorithm.

The difference between EDT and other distance transforms is the representation of the distance values. EDT does not just keep one value, but rather a two-element vector (u,v) to the closest object pixel. For SEDT, these components are allowed to be negative. The distance may then be computed as $(u^2 + v^2)^{1/2}$. Suggested masks are shown in section 2.4, Figure 12.

As mentioned above, parallel algorithms do not per se prevent sequential execution. Sequential algorithms, however, often but not always prevent parallel execution. In this paper we will investigate the possible execution of the sequential Euclidean distance transform on a parallel machine where the processors are arranged as a linear array. One such machine is PICAP3 [6], which is a coarse-grained linear SIMD array. It has a moderate number of powerful processors working in SIMD mode.

In this paper, we will *not* go into implementation details concerning low-level access in parallel architectures. Our goal is to outline algorithms that are possible to implement on certain architectures that conform to any of the models we use for parallel architectures. The models are simple, but that is necessary for making the results more generally useful.

2. Propagation of Euclidean distance values

2.1 Motivations and definitions

The SEDT versions presented by Danielsson and Ye can not be implemented in parallel with any gain from using several processors, since they do not allow more than one processor at a time to be active.

In this chapter, the propagation of distance values in Euclidean distance transforms is analyzed. Not only does the analysis give an explanation to why the original SEDT must use the scanning pattern in [3] (scanning each line back and forth) but it also points to alternatives that will lead to the final solution in our case.

Definition: By *object pixels* we refer to pixels with distance zero, that is, the pixels to which the distances (or vectors) in the distance map refer.

Definition: By *Voronoi polygon* we refer to a polygon enclosing all the pixels in the picture which has one and the same object pixel as the closest one.

Note that the division of a picture into Voronoi polygon is equivalent to a Voronoi diagram [2, 5, 11].

2.2 Voronoi Polygons

A Voronoi polygon is bounded by a number of straight lines. For each pair of object pixels the picture plane is cut into two halves my the mid-line, each half closer to one pixel than the other, as illustrated in Figure 6.

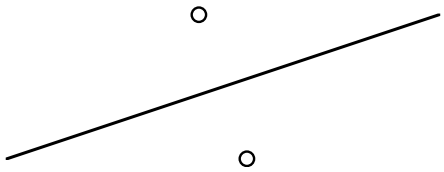


Figure 6. For each pair of object pixels, the picture plane is divided into two halves along a line.

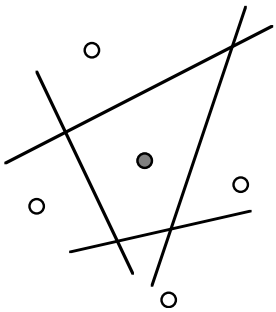


Figure 7. The intersection of all such half planes is the Voronoi polygon of the object pixel.

The Voronoi polygon of a certain object pixel is the intersection of all half planes on the object pixel's side of the midlines between itself and every other object pixel. See Figure 7.

Proposition: Voronoi polygons are convex.

This proposition is easily proven by induction.

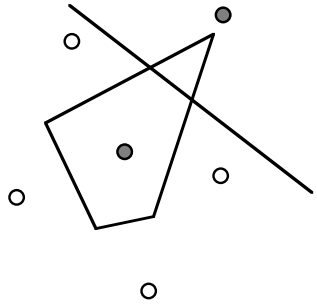


Figure 8. If a new object pixel is added, a part of the Voronoi polygon is cut off along a straight line.

The proposition infers that any pixel within a Voronoi polygon may be reached from any other pixel in the Voronoi polygon by a path along a straight line, which is completely within the Voronoi polygon.

2.3 Propagation

An important issue for distance transforms is the *propagation*. We must make sure that each pixel will get the minimal distance value possible, a value that has propagated from the closest object pixel. The distance values should propagate from all object pixels, and step by step fill out each Voronoi polygon with values from its proper object pixel.

The actual propagation paths used in a distance transformation depends on the algorithm as well as the picture. Many different propagation paths are possible. The question is, however, when the propagation works properly. The following theorem will answer this question.

Definition: By *complete propagation*, we say that the propagation of values is able to reach any pixel in the Voronoi polygon of a given pixel. This ensures that all pixels will get their correct value, provided that the distance values are exact, that is, Euclidean.

Theorem 1: If the algorithm supports propagation from any pixel position to infinity along any straight line, the propagation will be complete.

Proof: The propagation from one object pixel will always overwrite values from other object pixels while within that object pixel’s Voronoi polygon, and can not be overwritten by values from other object pixels. Thus, if propagation is possible from an object pixel to any other pixel in its Voronoi polygon without ever propagating through pixels outside its Voronoi polygon, the propagation can not be interrupted and will therefore be complete.

Since Voronoi polygons are always convex, it is always possible to draw a line from the object pixel to any other pixel in its Voronoi polygon. As long as the Voronoi polygon doesn't have corners with too acute angles (see below), a propagation path from an object pixel O to a pixel P can be guaranteed to exist only along the straight line through O and P.

Theorem 1 does not take into account the sampling of the image, which causes any propagation path to be non-linear. However, as long as the Voronoi polygons are not too narrow for the propagation path to fit, this is no problem. See section 2.5.

Theorem 1 is illustrated by the following example. The algorithm described in the introduction makes one picture scan to the right and downwards, and one scan to the left, upwards. Let us now consider this scanning pattern for a Euclidean distance transform. In a picture with one object pixel, the first scan will propagate over one quadrant, as in Figure 9a, and the second will propagate over the rest of the picture, as shown in Figure 9b.

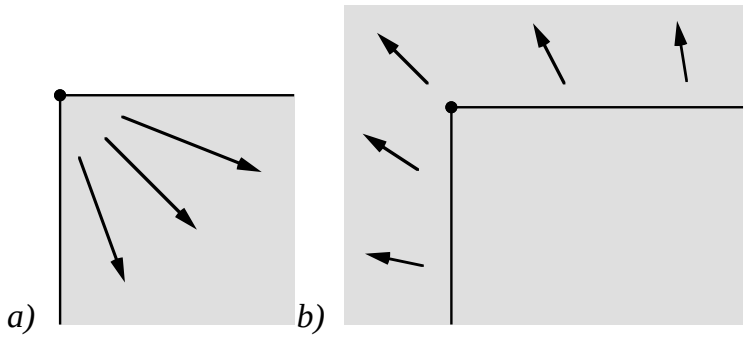


Figure 9. Propagation from one object pixel with a two-scan algorithm.

It may seem like the entire Voronoi polygon will be reached with this algorithm, since the propagation does reach any pixel in the picture in this example. However, as shown in Figure 10, it is not difficult to find a counterexample. If two object pixels are located in suitable positions, the first scan will result in a propagation shown in Figure 10a. In Figure 10b the result of the second scan is shown. The algorithm has failed.

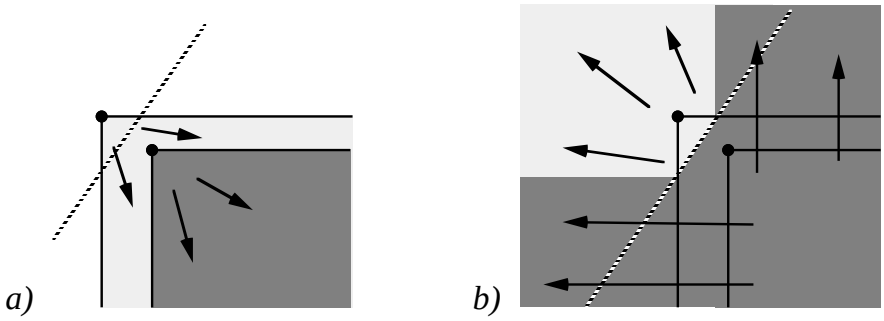


Figure 10. Propagation from two object pixels with a two-scan algorithm. The dotted line shows the border between the Voronoi polygons of the two object pixels.

These errors occur since the algorithm does not conform to Theorem 1. The upper object pixel in Figure 10 can not reach some parts of its Voronoi polygon directly, but only through areas in the Voronoi polygon of the other object pixel. These propagation paths were denied since the final, correct values reached these areas before they could be used for propagation on behalf of the upper object pixel.

2.4 Thin or elongated Voronoi polygons.

The hardest case for correct propagation is very thin or elongated Voronoi polygons. A thin part of a Voronoi polygon constrains the propagation path to a straight line. Incorrect algorithms like the one in the example above can also have problems with Voronoi polygons that are wide but very elongated, as illustrated in Figure 11. This motivates the formulation of Theorem 1.

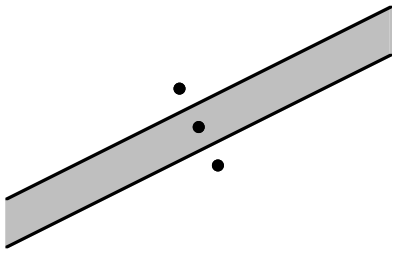


Figure 11. Voronoi polygons may have very elongated shape.

As an example, we will use it to verify the correctness of the sequential algorithms suggested by Danielsson and Ye.

We will primarily concentrate upon the 8-connective algorithms, 8SED or 8SSED (unsigned and signed, respectively). See Figure 12.

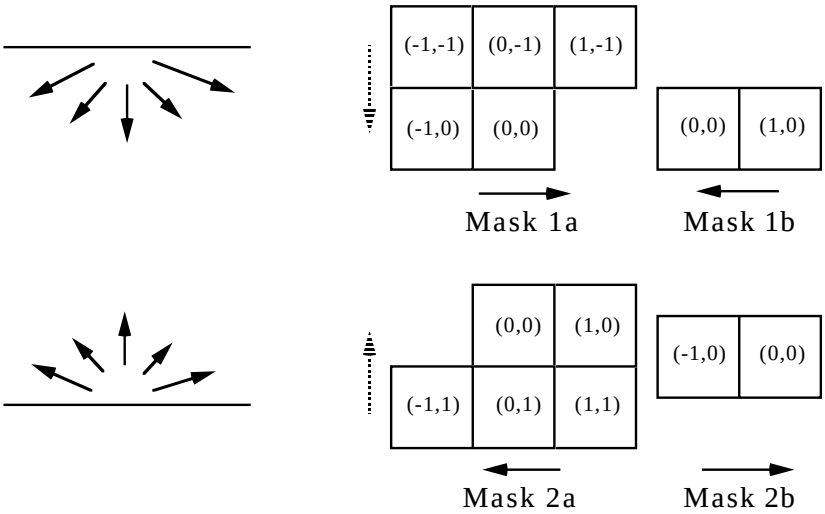


Figure 12. Masks for the original sequential EDT (8SED or 8SSED).

Each row is scanned both from left to right and from right to left. This is done first top-down and then bottom-up. This means that the value written to a pixel in one row scan may have propagated from any pixel on the same row and the previous row.

Mask 1a and 1b result in propagations in half of all possible directions. Mask 2a and 2b result in propagation in the remaining directions. Thus each picture scan will support propagation in all directions within an interval with the width π (180°). Together, the two scans support propagation in any direction, covering all possible elongated Voronoi polygons, as in figure 15.

In the following discussion, direction intervals have a central part. Therefore, we make the following definition:

Definition: A *direction interval* $DI[\alpha_1, \alpha_2]$ consists of all directions with angle α for which $\alpha_1 \leq \alpha + n2\pi \leq \alpha_2$ for any one integer value of n.

Most properties of direction intervals are intuitive, such as the meaning of the union of two direction intervals and the fact that all intervals $DI[\alpha_1 + n2\pi, \alpha_1 + n2\pi]$ for integer values of n are identical. Pie-chart symbols will also be used for illustrating direction intervals.

One scan of a sequential EDT supports propagation in a direction interval. In order for a scanning EDT to fulfill Theorem 1, the union of all direction intervals supported by all the scans of the algorithm must cover the entire direction space $DI[0, 2\pi]$.

As long as we consider the picture to be continuous, that is, we consider the propagation steps to be infinitely small, this implies that the sequential algorithms in [3, 11] fulfills Theorem 1.

2.5 The effect of using a picture with discrete pixels

So far, we have considered the picture to be continuous. The discrete nature of digital images will cause problems in narrow parts of Voronoi polygons, as shown in [3] and illustrated in Figure 13. This problem is a straight-forward violation of the condition in Theorem 1. An algorithm that is only performing neighborhood operations does not support propagation along any arbitrary thin line. For example, with 3·3 neighborhoods we do not support the propagation path shown in Figure 13.

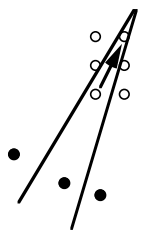


Figure 13. Propagation along any possible line may need to pass between discrete pixels.

The errors that arise from this effect are very few and scattered and are all much less than a pixel distance. Also, note that the magnitude of the errors does not increase with distance, which is the case for Chamfer distance transforms. As long as we want to use sequential, scanning algorithms, we must accept these errors to some extent. Since the errors are negligible, we may disregard the problem altogether for practical purposes.

Yamada [10] shows that the (on sequential computers, highly inefficient) parallel Euclidean distance transforms produce totally error-free distance maps. In parallel algorithms, propagations from different object pixels automatically form queues where several propagations advance in the same direction without overwriting each other. This idea is also exploited in the error-free sequential algorithm based on contour processing [8].

We conclude that though Theorem 1 provides a sufficient condition for an EDT to work in a broad sense, giving almost all pixels the correct value in a discrete image, we must take the sampling into account in some way to produce a totally error-free EDT, which demands other methods.

3. Implementation of sequential algorithms on parallel architectures

3.1 Hardware models

In the following, we will use two simple architecture models. We will primarily use a model that corresponds to PICAP3 and Picap 3.32, as described by Lindskog [6] and Eklund [4], respectively. PICAP3 is a linear array of SIMD processors, each connected to two neighbors and each with local, private memory. Figure 14 shows a simplified outline of the system.

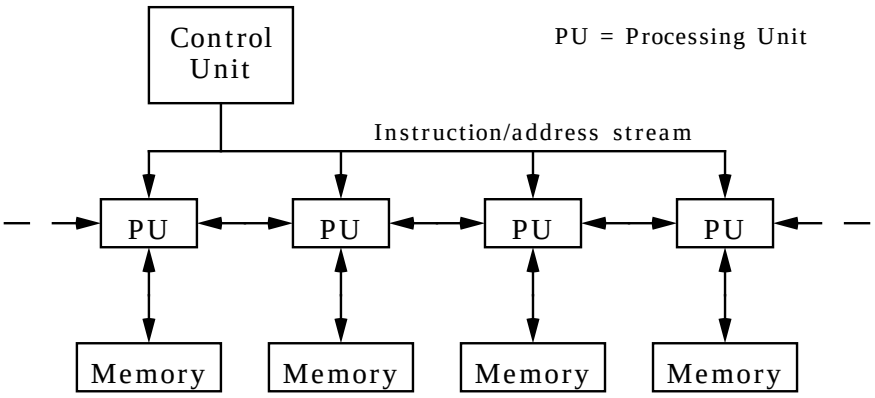


Figure 14. Linear SIMD array with separate memory for each processor and connections between neighbor processors. This is a crude model of how PICAP3 works.

It should be stressed that PICAP3 is a coarse-grained SIMD machine, that is, each processor is rather powerful. One particularly useful feature is the local address modification, that gives local addressing capabilities. In the following, we will take these kind of features for granted.

We will also mention the shared-memory SIMD model, also known as the PRAM (Parallel Random-Access Machine) model. In this case, a number of processors working in SIMD mode access the same memory space in parallel. See Figure 15. Such machines are divided into subclasses according to the access possibilities, whether two processors may access the same memory space simultaneously. On this subject, see for example Akl [1]. The PRAM is, however, a rather unrealistic architecture. A more realistic solution is sketched in Figure 16, where a network connects each processor to one memory block. Incidently, a vector processor like the Cray-1 can be considered a PRAM.

We will refer to these two models as the *PICAP3 model* and the *PRAM model*, respectively.

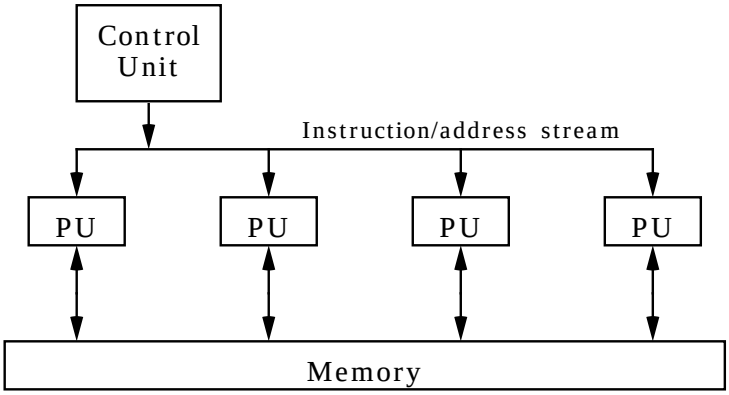


Figure 15. A SIMD machine with common memory for all processors, according to the PRAM model. This makes communication between processors trivial, but the model is rather unrealistic in practice, since it presumes parallel access to memory.

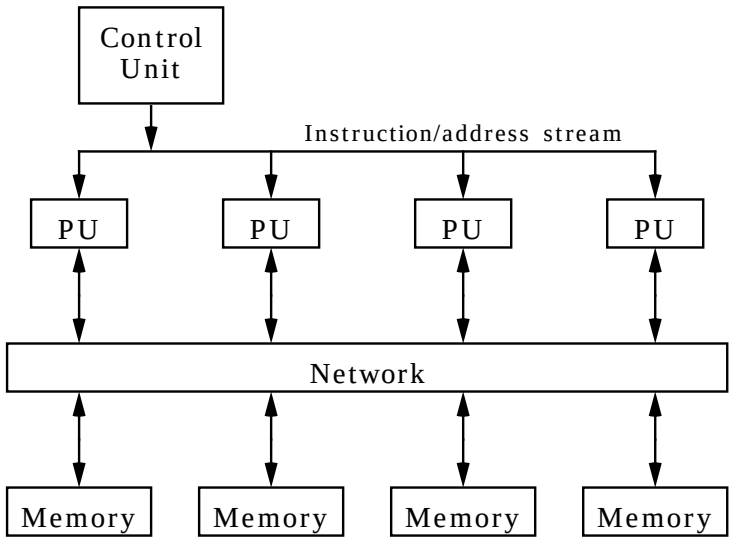


Figure 16. A more realistic architecture, related to PRAM. All processors can access all memory, but in a more limited fashion.

3.2 The partitioning of the image

Lindskog [6] suggests a number of different ways to partition the image. The simplest one is to partition it in a number of stripes, so each processor processes a continuous part of the image. This is the partitioning we will consider in the following discussion.

We will make the following assumptions. The number of columns in the image is divisible by the number of processors. Hence, each processor gets an equally large partition. The number of processors is not higher than the number of columns. If either of these assumptions are false, it can be helped by adding a number of extra columns and accepting some processors idle.

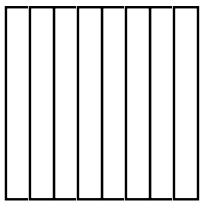


Figure 17. The picture is split in equal parts, giving each processor a number of consecutive columns.

With this architecture and partitioning, we may inspect existing distance transformation algorithms to see how they can be implemented.

Danielsson [3] suggests algorithms scanning each row in both directions in each picture scan. Let us call this kind of algorithm *double scanning*. As noted in section 2, they fulfill the demand of propagation in all directions (Theorem 1), but they can not be implemented in parallel, since the computation for each pixel depends on all previous computations.

Suppose we had a *single scanning* algorithm, that is, one that only scans each row in one direction, left or right, for each row in each picture scan. All non-Euclidean algorithms that are used in practice are single scanning, using two single scans, as exemplified in section 1. According to Montanari [7], two scans are sufficient for non-Euclidean metrics. Note that the masks typically used (as in Figure 3 and 21) access *at least one pixel on the same row as the center pixel*. This means that if we use such masks we must scan through each row sequentially.

Single scanning algorithms can be implemented in parallel. The following sections will discuss how this can be done.

3.3 Image scan with horizontal propagation front

The most straight-forward way to distribute the work among a number of processors working in each of a number of stripes, as suggested above, is to have them start simultaneously with the first pixel on the first row in their partition, and process the image in SIMD fashion.

This will lead to one limitation: many neighborhoods will not be useful. When processing a pixel, it may only address pixels in the previous line and beyond. Including pixels on the same line or the next will not add to the direction interval that is supported to infinity by the scan, so the increased processing by adding them is wasted.

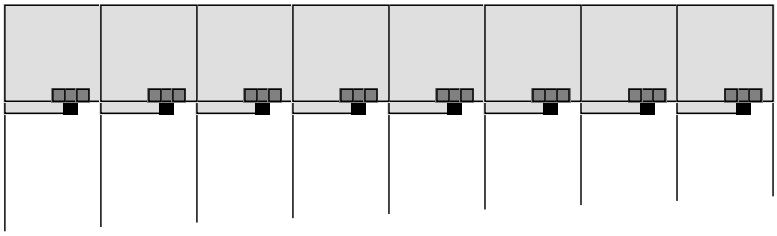


Figure 18. With all processors working on the same row in the image, we get a horizontal propagation front.

Figure 19. An typical neighborhood for use in a scan with horizontal propagation front, supporting propagation in the direction interval DI[-3π/4, -π/4].

Expressed in pseudo code, this comes out like:

Parallel implementation of sequential distance transformation algorithm, one picture scan only, horizontal propagation front

```
P processors (1..P)
K columns (1..K)
R rows (1..R)

All processors p∈{1..P} execute:

for r=1 to R
  for c=1 to K/P
     process(r ,c+(p-1)K/P)
```

In an EDT, it is of little interest to use larger neighborhoods than 3·3 ones. Hence, the mask in Figure 19 is a very reasonable one to use. Obviously, it supports a direction interval of π/2 (90°). As shown in the figure it is the interval DI[-3π/4, -π/4].

3.4 Image scan with sloping propagation front

An alternative to the horizontal propagation front is to have the processors processing different lines at any given moment. Two neighbor processors should process adjacent lines, as illustrated in Figure 20.

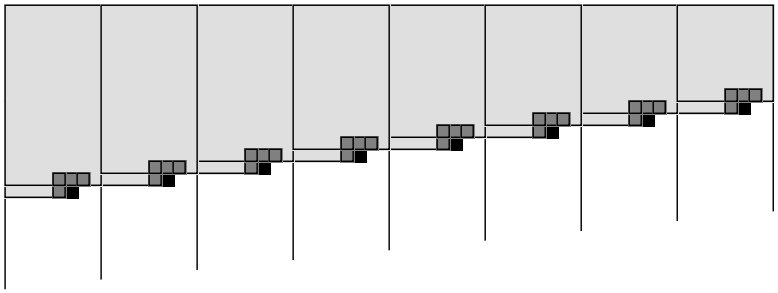


Figure 20. With a sloping propagation front, all processors are working on different rows in the image.

Figure 21. A typical neighborhood for use in a scan with sloping propagation front.

In the initial and final phase of the scan, some processors are idle. The loss of efficiency is increasing with increasing number of processors. Thereby, the gain from adding more processors is slightly lowered when using this kind of propagation.

Expressed in pseudo code, this comes out like:

Parallel implementation of sequential distance transformation algorithm, one picture scan only, sloping propagation front

```
P processors (1..P)
K columns (1..K)
R rows (1..R)

All processors p∈{1..P} execute:

for r=1 to R+P-1
  for c=1 to K/P
    if r-p≥0 and r-p<P
      process(r-p+1,c+(p-1)K/P)
```

Note that conditional processing capability, ability to make some processors idle, is needed here, a feature that is present in PICAP3.

In this case, a 3·3 neighborhood allows the slightly larger mask shown in Figure 21, which supports a direction interval of $3\pi/4$ (135°), DI[$-3\pi/4$, 0].

3.5 Combining several scans to complete distance transformations

This section concerns the propagation of the distance values. Here we should decide what masks to use and in which order to apply them. Our solution should fulfill Theorem 1 in section 2, and it must be single scanning according to section 3.2.

Suppose we use the masks used in the 8SSED algorithm, as shown in figure 12, but using four single scans instead of two double ones. If so, mask 1a will support propagation in directions within a 3π/4 (135°) interval, and so will mask 2a. The masks 1b and 2b, however, will not cover any direction interval at all, just a horizontal line. Therefore, we find the algorithm does no longer fulfill Theorem 1. The errors are also easily found by experiments. Figure 22 show the masks along with the direction intervals supported by each scan.

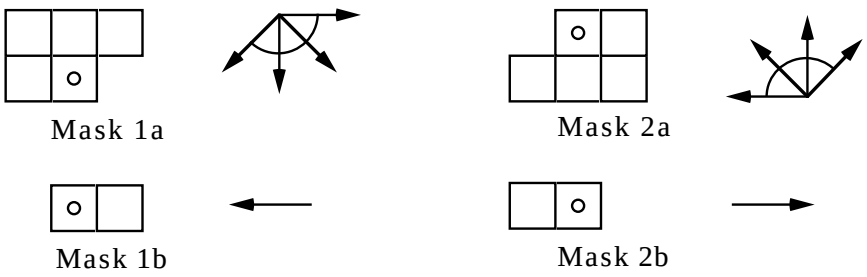


Figure 22. The 8SSED algorithm can not be separated into four separate scans.

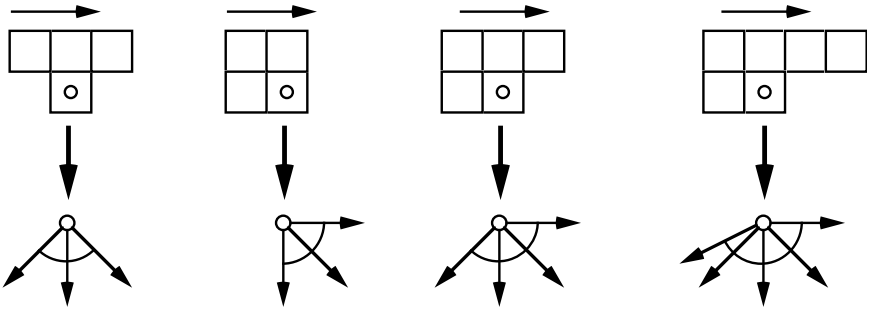


Figure 23. Some possible masks, resulting in propagations in all directions within different intervals, all less than π.

Since we must use single scans in the architecture chosen, the direction interval for one image scan is always less than π (180°), as illustrated in Figure 23. Therefore, we may never cover all angles with two scans, but we may do it using three scans, as shown in figure 24. This gives us a new 3-scan 8SSED algorithm.

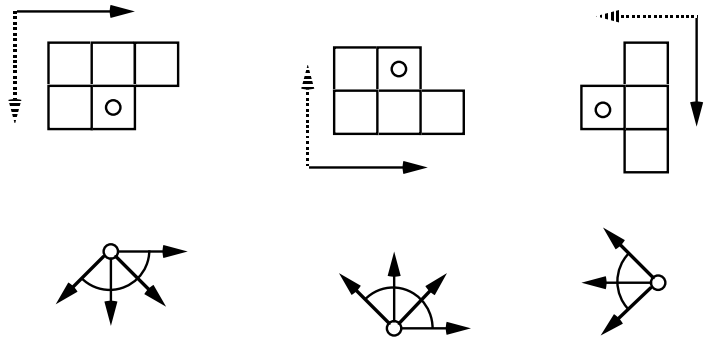


Figure 24. Masks for an algorithm using three single scans.

The masks define the 3-scan 8SSED algorithm. The first two masks scan row by row, but the third scans column by column. The third mask of Figure 24, however, forces one of the scans to be done in a different direction, scanning column by column. Such a vertical scan may be expressed in pseudo code as:

```
for column=...
  for row=...
```

while a horizontal scan looks like:

```
for row=...
  for column=...
```

The computer model of PICAP3 permits only horizontal scans. To do a vertical scan, we must transpose the picture, turning rows into columns and columns into rows. Transposing is a fairly time-consuming operation, compared to a distance transformation scan. Lindskog [6] calculated the time required for an N·N image on a PICAP3 with P processors to T_{tot} in (2).

$$T_{tot} = \frac{N^2}{2} \cdot \frac{1}{2} + \frac{25}{P} \quad (2)$$

This means that the operation takes $O(N^2)$ time when P is high, while a distance transformation scan takes $O(N^2/P)$. Hence, on a PICAP3 architecture, we should avoid transpositions during distance mapping, if possible, and the algorithm suggested by Figure 24 is not suitable. On a PRAM architecture, transposition is not needed, so the three-scan version is perfectly suitable in this case as well as in the case of a single processor computer.

Figure 25 shows a set of masks which together cover the full direction space of 2π. They define the four scan 8SSED/SIMD algorithm, a new version of 8SSED that can be used in a linear SIMD array.

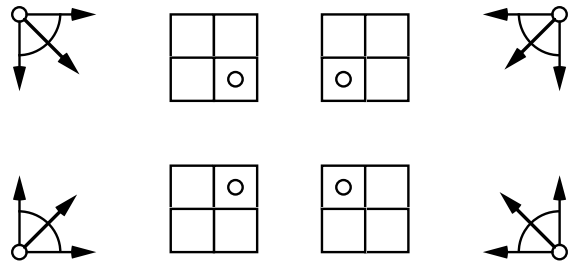


Figure 25. A solution using four single scans. This mask set is preferable using an architecture like PICAP3, since the image does not need to be transposed. This defines the 8SSED/SIMD algorithm.

A PRAM architecture, where transposition is not necessary (or at least can be done in shorter time than one distance transformation scan), we may use a four-scan solution with masks that use all processors at full speed throughout the operation. These masks are shown in Figure 26, which defines the 4H-8SSED/SIMD algorithm (8SSED/SIMD with 4 horizontal scans).

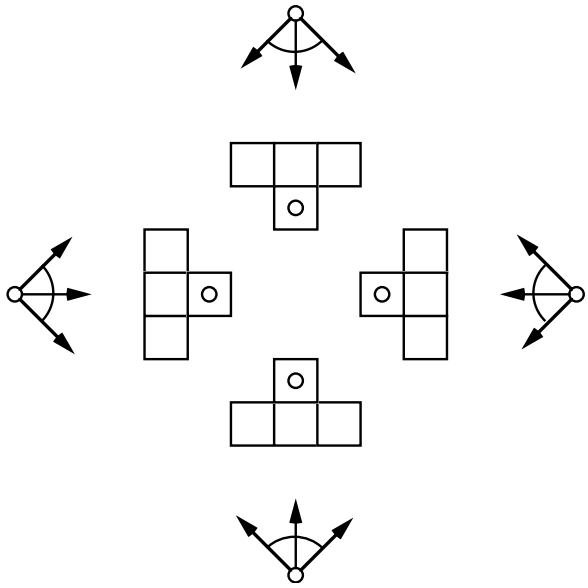


Figure 26. A solution using four single scans. This mask set is not useful with the PICAP3 model, but for other architectures like the PRAM. This defines the 4H-8SSED/SIMD algorithm.

All these three mask sets are single-scanning versions of the 8SSED algorithm as defined by Danielsson [3]. Here give them the generic name 8SSED/SIMD algorithms *(8SSED suitable for SIMD-architectures)*, since they all can be implemented in parallel on linear SIMD arrays.

A relative to the 8SSED/SIMD of Figure 25 should also be recognized. This is the 4SSED/SIMD, defined by the masks of Figure 27, which is similar to the 8SSED/SIMD but with one pixel less in each mask. This feature gives higher speed but a larger number of pixels with small errors.

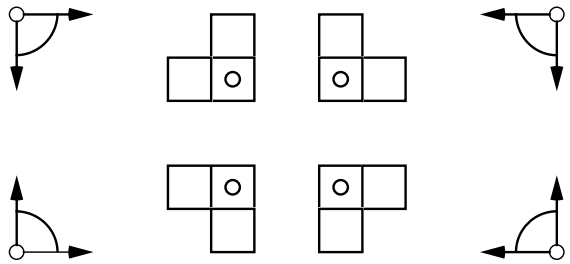


Figure 27. Masks for the 4SSED/SIMD algorithm, a 4-neighbor version of the 8SSED/SIMD. This version is faster but the resulting distance map has more errors.

The algorithms for applying two of the masks employed in Figures 25-27 are described below in pseudo code. Algorithms for the others can easily be obtained by changing scan and mask directions. The two masks described are the ones in Figure 28.

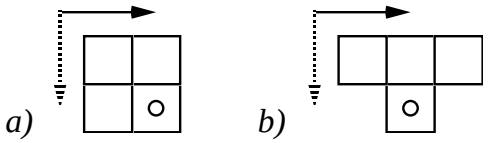


Figure 28. The masks and scanning directions for the two scans described in the pseudo code. a) Sloping propagation front. b) Horizontal propagation front.

8SSED/SIMD in parallel, examples of scans

```
Working on a picture with
R rows (1..R)
K columns (1..K)
using P processors (1..P)

Start:
L(i,j) := (0,0) for all object pixels
L(i,j) := (Z,Z) otherwise

All processors p∈{1..P} execute the algorithms below.

One picture scan, sloping propagation front:
for row := 1..R+P-1
for column := 1..K/P
if row-p≥0 and row-p<P
process2(row-p+1,column+(p-1)K/P)

using the procedure
process2(i,j)
L(i,j) := min{
        L(i,j)
        L(i-1,j) + (-1,0)
        L(i-1,j-1) + (-1,-1)
        L(i,j-1) + (0,-1)
```

```
    }

One picture scan, horizontal propagation front:
for row := 1..R
  for column := 1..K/P
    process1(row,column+(p-1)K/P)

using the procedure
process1(i,j)
L(i,j) := min{
      L(i,j)
      L(i-1,j-1) + (-1,-1)
      L(i,j-1) + (0,-1)
      L(i+1,j-1) + (+1,-1)
      }
```

The code above does not describe the actual computation of distance values, i.e. evaluation of the expressions of the type L(i,j) := min(...). For each pixel (i,j) we hold a two-component vector (u,v), pointing to the closest object pixel. The distance L is the length of the vector, that is $L = (u^2+v^2)^{1/2}$. For each neighbor in the mask we take its vector and modify it to point from the center pixel to the object pixel.

For an example, take the neighbor at (i-1,j). If this position contains the vector (u',v'), pointing to an object pixel, the vector (u'-1,v') points from the center pixel to the same object pixel. If $((u -1)^2 + v^2)^{1/2} < (u^2 + v^2)^{1/2}$, then the vector (u,v) in the center pixel can be replaced by (u'-1, v'). It may seem costly to have to compute several square roots for each processed pixel. In reality, we implement the algorithm avoiding both square roots and multiplications. This can be done using a lookup table, as suggested by Ye [11].

A potential new vector and distance value should be computed for each neighbor in the mask. The vector corresponding to the lowest of these distance values is put in the center pixel.

3.6 Comparison of the speed of the resulting algorithms

Suppose that we have a linear SIMD computer with P processors, and we want to make a distance map of an image of N·N pixels, and the time required to process one neighborhood is 1, what running time does an image scan require?

We have two cases: horizontal propagation front and sloping propagation front. With horizontal propagation front, all processors are active during the whole operation, while for a sloping propagation front, some processors are idle under the initial and final phases. This gives the running times (3) and (4) for a single scan.

With horizontal propagation front: N^2/P (3)

With sloping propagation front: $N\cdot(N+P-1)/P$ (4)

This means that both kinds of scans take approximately the same time when $P << N$. For $P \approx N$, a scan with horizontal propagation front takes half the time of a scan with sloping propagation front. For the complete algorithms, (3) and (4) together with Figures 24, 25 and 26 give the total running times (5) (6) (7).

8SSED/SIMD: $4 \cdot N \cdot (N+P-1)/P$ (5)

4H-8SSED/SIMD: $4 \cdot N^2/P$ (6)

3-scan 8SSED: $2 \cdot N \cdot (N+P-1)/P + N^2/P = 3N^2/P + 2N(P-1)/P$ (7)

For a PICAP3 architecture, the 8SSED/SIMD and 4SSED/SIMD are the only reasonable choices. For a PRAM, the 3-scan 8SSED is preferable with a small number of processors, while the 4H-8SSED/SIMD is faster with many processors, i.e. $P \approx N$.

Finally, let us compare the above results to the complexity and performance of equivalent algorithms on totally different architectures. Algorithms have been presented for single processor computers [3, 11] and for 2-dimensional SIMD meshes [3, 10]. With one processor (0-dimensional architecture) a scanning algorithm has a running time $\in O(N^2)$. With a 2D mesh, we can only use a parallel algorithm, which will have a running time $\in O(N)$ time. This should be compared to the algorithms in this paper, that have running times $\in O(N)$ on a 1-D architecture.

Note that some architectures can be configured both as a mesh or a linear SIMD, which gives them the option of using either kind of algorithm. See further below.

The *cost* of an algorithm is, according to Akl [1], defined to be

$$\text{cost} = \text{number of processors} \cdot \text{running time}$$

That gives the following table, where the cost should be as low as possible:

Processors	Time	Cost
$O(N^0)$ (Single PU)	$O(N^2)$	$O(N^2)$
$O(N^1)$ (Linear SIMD)	$O(N^1)$	$O(N^2)$
$O(N^2)$ (Mesh)	$O(N^1)$	$O(N^3)$

Hence, for this kind of algorithms, a linear SIMD array gives us a speed comparable to that of a 2-dimensional SIMD array with N times less processors.

For architectures that allow both mesh and linear organization, we must rather compare the performance for a given number of processors P. In this case, each processor in a Mesh architecture will process all pixels in a $N/\sqrt{P} \cdot N/\sqrt{P}$ square. For each iteration, each processor must process N^2/P pixels. Still assuming the number of iterations to grow with image size, O(N), the time is in $O(N^2/P \cdot N) = O(N^3/P)$.

With a number of processors far smaller than the number of pixels, we should also consider a hybrid algorithm, an algorithm where each processor uses a sequential algorithm when processing the $N/\sqrt{P} \cdot N/\sqrt{P}$ pixels area it is responsible for.

In this case, one iteration performs a full DT in the area allotted for each processor, taking into account changes in the border pixels caused by the previous iteration. Since each square is $N/\sqrt{P}$ by $N/\sqrt{P}$ pixels, $O(N/\sqrt{P})$ steps of propagation is performed for each iteration. Thus, only $O(\sqrt{P})$ iterations are needed. This improves the time to $O(N^3/(P \cdot \sqrt{P})) = O((N/\sqrt{P})^3)$.

We get the following table.

Architecture	Processors	Time	Cost
Single PU	1	$O(N^2)$	$O(N^2)$
Linear SIMD	P	$O(N^2/P)$	$O(N^2)$
Mesh, hybrid	P	$O((N/\sqrt{P})^3)$	$O(N^3/\sqrt{P})$
Mesh, parallel	P	$O(N^3/P)$	$O(N^3)$

The Linear SIMD architecture/organization comes out as a clear winner, with higher speed for the same number of processors.

4. Conclusions

We have investigated the possibility to implement the Euclidean Distance Transform on parallel architectures, and described ways to implement scanning algorithms on a 1-dimensional SIMD array of the PICAP3 type or on a PRAM architecture. Four new algorithms were suggested:

- 3-scan 8SSED, suitable for both single-processor computers and PRAMs with a small number of processors. It is less suited for PICAP3-style architectures, since the image needs to be transposed.
- 4H-8SSED/SIMD, suitable for PRAM architectures with a large number of processors.
- 8SSED/SIMD, suitable for PICAP3-style architectures.
- 4SSED/SIMD, a fast, low-quality version of 8SSED/SIMD.

The algorithms are supported by a theorem about the propagation paths for Euclidean distance propagation, saying that we must support propagation along straight lines in all directions. All algorithms were derived from this theorem.

These scanning algorithms, which can be implemented on 1D parallel archiectures, seem to have a very favorable performance/cost ratio compared to parallel distance transform algorithms.

References

[1] S.G. Akl, *The Design and Analysis of Parallel Algorithms*, Prentice Hall, Eaglewood Cliffs, New Jersey, 1989.

[2] G. Borgefors, "Distance Transformations in Digital Images", *Computer Vision, Graphics and Image Processing* 34, No.3, 1986, pp 344-371.

[3] P.E. Danielsson, "Euclidean Distance Mapping", *Computer Graphics and Image Processing* 14, 1980, pp 227-248.

[4] S. Eklund, "Modellering och simulering av parallelldatorsystem - En studie av ELLA och Picap 3.32", *Linköping Studies in Science and Technology, Thesis*, No. 247.

[5] L. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams", *ACM Transactions on Graphics*, Vol. 4, No 2, April 1985, pp 74-123.

[6] B. Lindskog, "PICAP 3, An SIMD architecture for multi-dimensional signal processing", *Linköping Studies in Science and Technology, Dissertations*, No. 176.

[7] U. Montanari, "A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance", *Journal of the ACM*, vol 15 No 4, October 1968, pp 600-624.

[8] I. Ragnemalm, "Contour processing distance transforms", *Progress in Image Analysis and Processing*, World Scientific, Singapore, 1990, pp 204-212.

[9] A. Rosenfeld, J.L. Pfaltz, "Sequential Operations in Digital Picture Processing", *Journal of the ACM* 13, No 4, 1966, pp 471-494.

[10] H. Yamada, "Complete Euclidean Distance Transformation By Parallel Operation", *Proceedings, 7:th International Conference on Pattern Recognition*, 1984, pp 69-71.

[11] Q.Z. Ye, "The Signed Euclidean Distance Transform and Its Applications", *Proceedings, 9:th International Conference on Pattern Recognition*, 1988, pp 495-499.

[12] B.J.H. Verwer, P.W. Verbeek and S.T. Dekker, "An Efficient Uniform Cost Algorithm Applied to Distance Transforms", *IEEE Trans. on Pattern Analysis and Machine Intelligence 11*, 1989, pp 425-429.

Appendix: The Quick-And-Dirty SSED algorithm

The following results were cut from the above paper in order to shorten it, making it more suitable for publication. These results are minor, mostly included as some marginally interesting side notes.

For high precision, we can use a 8-neighbor EDT like the 3-scan EDT or the 8SSED/SIMD. With lower demand for precision, we can use the 4SSED/SIMD. However, it is possible to go even further in sacrificing precision for speed. Figure 24 suggests that we could cover most of the possible directions using only two scans. However, since it only covers *most* directions, some directions are no longer supported, and we do no longer fulfill Theorem 1.

The result is a faster and simpler algorithm with larger and much more frequent errors. The algorithm, in the following text referred to as *Quick-And-Dirty SSED*, uses the masks shown in the figure below. This variant is exactly twice as fast as 4SSED/SIMD, since its masks are of the same size, but it uses only two scans. The masks for these two scans are shown in Figure 29.

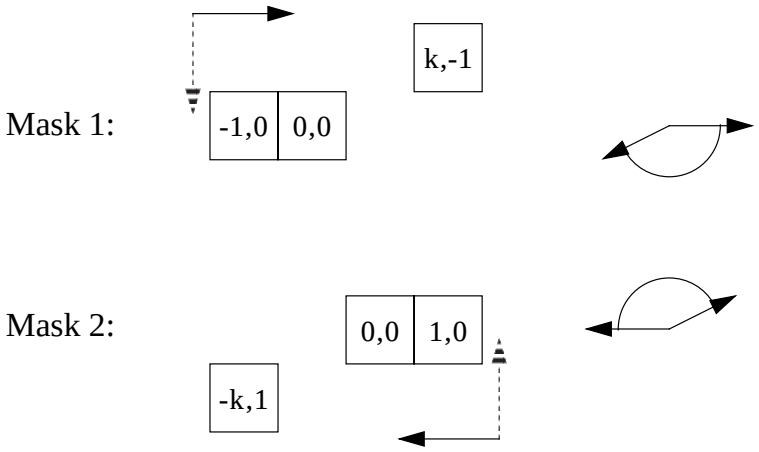


Figure 29. Masks for Quick-And-Dirty SSED

Note the variable **k**. By choosing different k values, we get different behavior. A small k, like 1 or 2, will cause more errors in large areas, since the supported direction intervals (shown in Figure 29) will cover a smaller part of the direction space. We name this kind off error *missing direction error* (MDE).

Figure 30 illustrates the MDE errors. In the Figure, the propagation from the upper of two object pixels (small shaded circles) is supported in the interval between the arrows from the object pixel. In the second scan, the propagation from the lower object pixel will overwrite the pointers to the upper object pixel before they can propagate in to the shaded area. Thus, the shaded area will refer to the lower object pixel, though the upper one is closer.

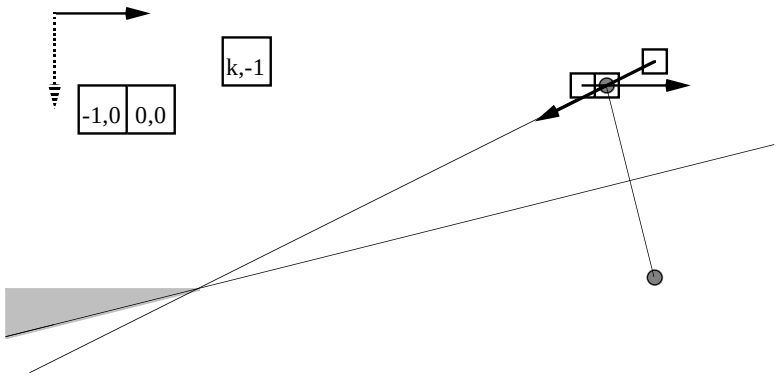


Figure 30. Two object pixels in a configuration where the shaded areas will refer to the wrong object pixel.

Using a larger k, the MDE errors will be reduced, but instead more errors in discrete pixels will occur. For large k values, this problem will not only cause errors in single pixels, but rather in small areas. This is caused by the low density of the masks, why we name it *low density error* (LDE).

Like the occasional errors occurring in 4SED and 8SED, LDE errors occur in narrow parts of the Voronoi polygons, typically in corners with acute angle. As long as the *center pixel and at least one neighbor* of the mask can fit within the Voronoi polygon, we will get correct results. In the narrow parts of Voronoi polygons where this is not possible, propagation is not supported and we will get errors. This is illustrated in Figure 31.

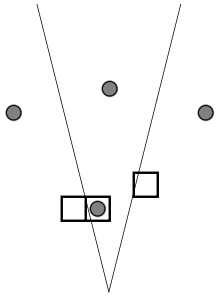


Figure 31. Low density errors occur when a mask can not fit in a narrow part of a Voronoi polygon, so none of the mask members is in the same Voronoi polygon as the center pixel.

The LDE can be reduced by using one more pixel in each mask, which gives us the *Less-Quick-And-Dirty-SSED*. We should then have one pixel with a large k value, but also a pixel with a small k, zero or possibly 1. See Figure 32. Experiments show very few errors with this algorithm, but the speed is not double the speed of 4SSED/SIMD any more, but instead double the speed of 8SSED/SIMD. The improvement compared to 4SSED/SIMD is rather about 50-60%.

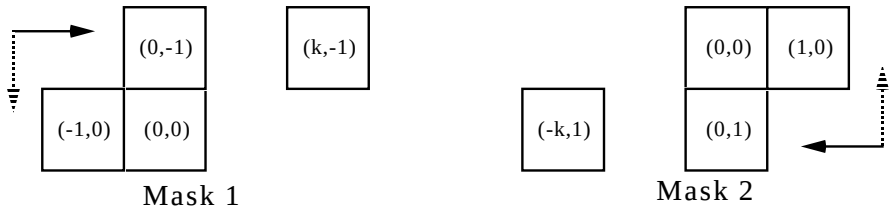


Figure 32. Masks for Less-Quick-And-Dirty SSED, a variant which is slower, but with much less LDE errors.

Both approaches will cause a lot more errors than EDT algorithms satisfying theorem 1. It is, however, more appropriate to compare them to more approximative DTs like Chamfer 3-4 or Chamfer 5-7-11.

The maximum error caused by using two scans instead of four

There are, as mentioned above, two kinds of errors that may occur in a distance map produced by a Quick-And-Dirty EDT. MDE, the one caused by not being able to propagate in all directions, is impossible to avoid completely by other means than using more than two scans. The LDE's can be reduced by adding more pixels to the masks. In this section, we will analyze what the worst case for MDE is for different versions of Q-A-D SSED.

Assume we have an image with two object pixels, placed in relative positions so that errors will occur. Errors will then occur only in certain areas, as shown in Figure 30.

Close to the real Voronoi polygon border, the distance to the object pixels are almost equal. They will differ more the farther we go from the vicinity border. The largest errors will occur along the horizontal line, where the difference between the distances to A and B is largest.

Let us define a number of variables, in Figure 33. The two object pixels, named A and B, are located at the distance b from each other, and at the angle φ. In cartesian coordinates, the distance may be expressed as Δx and Δy.

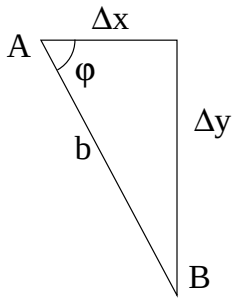


Figure 33. Two object pixels A and B, and the symbols for their relative position.

From the geometry of the triangle we have $b = \sqrt{\Delta x^2 + \Delta y^2}$ and $\varphi = \operatorname{atan}\frac{\Delta y}{\Delta x}$.

The k value, defined by the choice of mask, gives the angles q and p, shown in Figure 34, showing within what angles propagation is supported. We define:

$$q = \operatorname{atan} k$$

$$p = q + \frac{\pi}{2} - \varphi$$

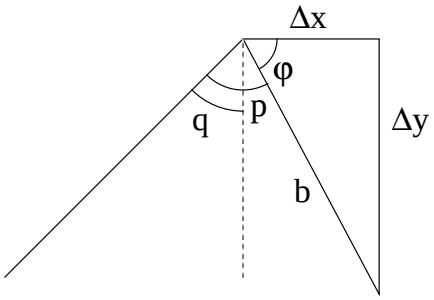


Figure 34. Propagation directions in one scan is possible within an φ+p angle interval.

The errors will occur beyond the point where the vicinity border and the p slope intersects. We name this point F. The distance to F from any of the object pixels A and B (equal distance since we follow the Voronoi edge) is D in Figure 35.

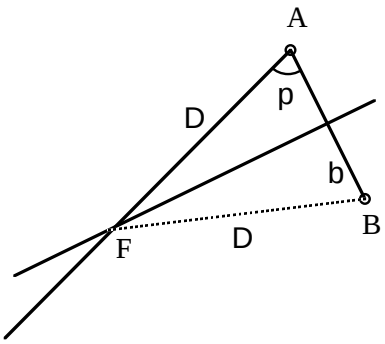


Figure 35. The point F is the point beyond which errors may occur. The distance from either of A or B is D.

The Voronoi polygon border is normal to a line passing through A and B. The triangle formed by ABF is equilateral. This give us the distance D:

$$D = \frac{b}{2\cos p}$$

As noted above, the maximum errors will occur along the horizontal line, the upper edge of the area with erroneous values, starting at F. Let ε be the distance to F along this line. The distance to a pixel along the line from A is L_1, and from B is L_2, as shown in Figure 36. Since the correct distance value is L_1, but the erroneous pixels will get the distance value L_2.

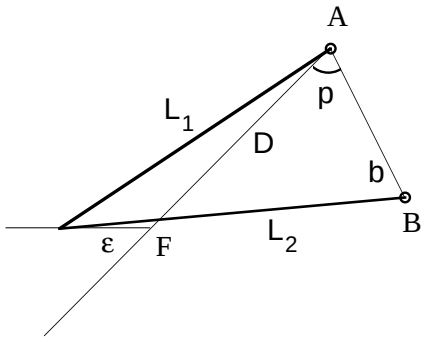


Figure 36. The distance L_1 and L_2 to a point, located at the distance ε from the F point along the horizontal line.

We may then find these distances as

$$L_1 = \sqrt{(D\sin q + \varepsilon)^2 + (D\cos q)^2}$$

$$L_2 = \sqrt{(D\sin q + \varepsilon + \Delta x)^2 + (D\cos q - \Delta y)^2}$$

Note that errors only occur when $q < \varphi < \frac{\pi}{2}$. We define the absolute error as $E_{abs} = L_2 - L_1$. We have an absolute maximum for this error, depending on the positions of A and B. For very large ε, the maximum error approaches Δx, but there is a local maximum for some ε.

There is no maximum absolute error. The absolute maximum error depends on the pixel locations and the image size. However, the relative error is maximized. We define the relative error as

$$E_{rel} = \frac{L_2 - L_1}{L_1} = \frac{L_2}{L_1} - 1.$$

This is a function of φ, b, q and ε. By numerical methods we have found a the maximum given some values of k. The maximum relative error was found to be independent of b, as one would expect. For all cases, the maximum error was found at

$$\varphi_{max} = \frac{\pi}{4} + \frac{q}{2}$$

The results are shown in Table 1.

Table 1:

k	maximum relative error (%)
0	41.4
1	8.2
2	2.75
3	1.31
4	0.76
5	0.49
6	0.34
8	0.19
10	0.12
15	0.055

The maximum errors drops rapidly and has reasonable values at k=2 and up. However, we should beware of using large k for the Quick-And-Dirty SSED, since the LDEs will increase. The Less-Quick-And-Dirty SSED allows larger k values, but even there the errors will get worse for large k.

The errors should be compared to the Chamfer algorithms [4]. The two recommended Chamfer algorithms are the Chamfer 3-4, using a 3·3 neighborhood, and the Chamfer 5-7-11, using a 5·5 neighborhood. According to [4], they have maximum errors of about 8% and 2% respectively.

In [3] it is shown that we should expect the Quick-And-Dirty SSED to be slightly slower than Chamfer 3-4, and the Less-Quick-And-Dirty SSED to be even a little slower, but still faster than the Chamfer 5-7-11. In that context we should demand the QAD SSED to be more accurate than the Chamfer 3-4 but not necessarily more accurate than the Chamfer 5-7-11, which the LQAD SSED should.

This says that QAD SSED should have a k value of at least 2, whereby it becomes far better than the Chamfer 3-4. The LQAD SSED should also have a k of at least 2, but preferably 3. If so, the two QAD SSED should are superior to the non-Euclidean distance transforms. They have less maximum errors for equal speed, and the errors only occurs in small areas. In most of the picture, the distance values are Euclidean. Of course, the accuracy is still inferior to the ordinary Euclidean transforms.

The errors caused by using incomplete masks

The numbers in Table 1 look very attractive, but note that we have not included the LDE in the analysis above. We will not go into depth on the maximum error possible, but rather note that the errors can be rather big, particularly for the QAD SSED. For QAD SSED, we can easily find cases where the absolute error is close to k. LQAD SSED behaves

much better in this respect. For LQAD SSED with k=3, the biggest LDE error we have found is 0.6 pixel distances, at distance $\sqrt{2}$. Even then, the relative LDE error is quite big due to the small distance where the error occurs. For a large enough distance, the LDE errors get smaller than the MDE errors, and Table 1 will be valid.

The big LDE errors make the QAD SSED rather useless. The LQAD SSED is somewhat more interesting, but the existence of efficient ordered propagation algorithms for EDT (paper #2 and later work, developed after the QAD algorithms were invented) makes even that a mere curiosity.

Neighborhoods for Distance Transformations using Ordered Propagation

CVGIP: Image Understanding 56, 1992.
Short version: “Contour Processing Distance Transforms”, in: Cantoni et. al. eds, *Progress in Image Analysis and Processing*, World Scientific, Singapore, 1990.

Neighborhoods for Distance Transformations using Ordered Propagation

Ingemar Ragnemalm

Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden

Abstract:

This paper presents a fast method for creating distance maps using ordered propagation techniques, where only the propagation front is processed in any stage of the operation. In this paper, we place the emphasis on the size of the neighborhoods used, the number of pixels that must be inspected for each pixel that is processed. We have developed algorithms for both non-Euclidean and Euclidean metrics where only two neighbors need to be inspected per pixel. For Euclidean distance maps, a version is proposed that is totally error-free.

1. Introduction

A distance map is an image where each pixel tells the distance to the closest set pixel (object pixel) in a corresponding binary image. A number of applications exist that use distance maps for extracting information from an image. [7]

In ordinary distance maps, the distance values are just that, a single value telling the distance [1, 3, 8, 11, 12]. Such distance maps typically use non-Euclidean metrics like the City Block distance, Chessboard distance or the more accurate Chamfer distance (pseudo-Euclidean) metrics.

An alternative is the *Euclidean distance map*, where we use a two-component vector (three for 3D images, etc.) that tells the distance to the closest object pixel along each coordinate axis [2, 5, 7, 10, 15]. An example is shown in Figure 1. For a vector (x,y) the distance is $(x^2 + y^2)^{1/2}$. As the name implies, this distance is the exact Euclidean distance. If this vector tells the exact position of the object pixel instead of just telling the distances, the distance map is said to be *signed*. In the following, we consider all Euclidean distance maps to be signed. The extra information supplied by Signed Euclidean distance maps give a wider range of applications than ordinary distance maps or unsigned Euclidean distance maps. [7, 15].

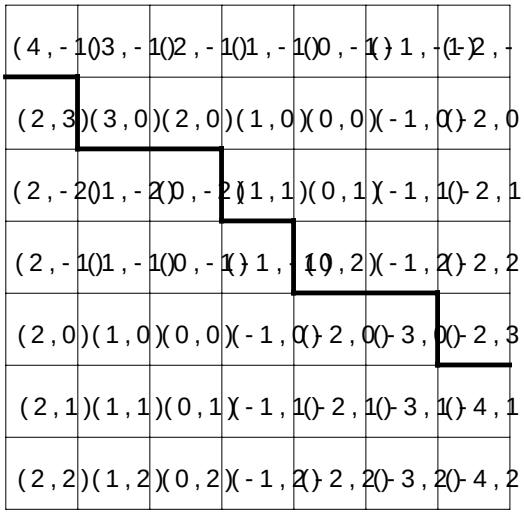


Figure 1. A signed Euclidean distance map generated by applying the Signed Euclidean Distance Transform on a binary image with two feature pixels.

A distance map is generated with a *distance transform* (DT), while a Euclidean distance map is generated by a *Euclidean distance transform* (EDT). The term *distance transform* is often also used for the distance map itself, the output from the algorithm.

The normal method to generate distance maps of either kind is to use a *recursive*, raster scanning algorithm that scans the image twice (or more) [1, 2, 5, 8, 16]. When a pixel is processed (referred to as the *center pixel*) its distance value is compared to the distance value of a number of neighbors (modified to refer to the center pixel) and the lowest value found is written into the center pixel. This will cause the distance values (or vectors) to propagate from each object pixel in the direction of the scan.

We could generate distance maps using a *parallel* algorithm, where all pixels are processed in each iteration [2, 9]. If so, we should iterate until there are no changes in the distance map during a whole iteration. This will, however, require a lot more processing. It has a complexity of $O(n^3)$ instead of $O(n^2)$ for a 2D image. This makes parallel algorithms truly useful only on massively parallel systems.

This paper discuss a third possibility, namely ordered propagation (a.k.a. contour processing). Piper and Granum [4], Verwer et al [11, 12] and Vincent [13] have described algorithms using related techniques. Van Vliet and Verwer [14] describe a related algorithm for binary neighborhood operations. This paper will mainly focus upon the possibility to reduce the number of neighbors inspected per pixel.

It should be noted that Euclidean distance maps generally are not totally error-free, as pointed out by Danielsson [2]. The raster scanning EDTs produce distance maps where single pixels may occur that have an incorrect value. These errors are always less than one pixel distance. Yamada [9] has, however, shown that certain algorithms do produce error-free distance maps. See further section 5.

While Piper and Granum [4] consider domains of many different shapes, we consider only convex domains here, particularly the usual 2-dimensional rectangular image array. The extension of the algorithms into three dimensions is rather straight-forward. The case of non-convex domains, constrained DTs, is more complicated. Constrained DTs using pseudo-Euclidean (i.e. Chamfer) metrics do not benefit from using directional information as used extensively in this paper, but should rather use complete neighborhoods [4, 12]. The Euclidean DT, however, must use directional information to work in a non-convex domain. This is demonstrated by the constrained EDT as described in [7], using ordered propagation and directed masks.

2. Ordered Propagation Algorithms

In both parallel and recursive algorithms, as described above, much processing power is wasted in processing pixels that already have received their correct distance values or that have not yet been reached by propagation from the object pixels. Each pixel is thereby processed several times. Ideally, each pixel should only be processed *once*. The processing of a pixel is only truly useful when the pixel receives its final value. We can get very close to this goal by using ordered propagation.

We will now describe our algorithm for DTs using ordered propagation techniques. First, we make the following definition.

Definition: In any stage of the transform, only a limited set of pixels is possible to process with any benefit. We define this set as the *Contour Set*.

In the initial phase of a DT, only the neighbors of the object pixels can be changed. Therefore, the set of object pixels on edges of objects is the initial Contour Set. In later stages, the Contour Set holds pointers to pixels on the propagation front.

In an ordered propagation algorithm, the processing of a pixel is different from the usual procedure for raster-scanning methods. When a pixel is processed in a raster-scanning algorithm, the neighbors are inspected, their distance values modified to refer to the center pixel, and the center pixel is set to the lowest distance value. Verwer et. al. [12] use the term *read formalism* for this procedure.

The alternative, used in our algorithms, is *write formalism*. In this case, the distance value (vector) of the center pixel is modified to refer to each of the neighbor pixels, and the neighbors are updated if the new distance value is lower. See [12] for definitions of write and read formalism.

The set of neighbors that are inspected in either case are often called the *mask* for the processing of the pixel.

Suppose we process an arbitrary pixel in the Contour Set. It may update some neighbor pixels. The neighbors of the updated neighbors must then be marked for future processing. This is done by adding pointers to the updated pixels to the Contour Set. We also remove the processed pixel from the Contour Set. We will compute the DT simply by repeating this until the Contour Set is empty.

The algorithm described above is expressed in the following procedure:

1: Put pointers to all the object pixels into the Contour Set. Set the distance value of all object pixels to zero and the distance value of all background pixels to a sufficiently high value.
2: Choose a member of the Contour Set and remove it from the set. This member points to a pixel that will be processed. We name it the Center Pixel.
3: For each of the pixels in a neighborhood of the pixel:
3.1: Modify the distance value of the Center Pixel to refer to the neighbor instead. (In non-Euclidean algorithms, this is done by simply adding the distance from the Center Pixel to the neighbor.)
3.2: If this modified distance is less than the distance value stored in the neighbor:
3.2.1:Store the distance value into the neighbor.
3.2.2:Put a pointer to the neighbor into the Contour Set.
4: If the Contour Set is not empty, repeat from step number 2.

This algorithm is written in a general manner, and leaves a lot of options. We have said nothing about which pixel to choose in step 2 or what neighborhood we will use in step 3.

Piper and Granum [4] suggest a queue, putting the modified pixels in the end of the queue and taking new center pixels from the front. We could use a sentinel to separate different iterations, which is needed in some applications. We may also use two different queues, one for the current and one for the next iteration.

However, this approach may cause many pixels to be updated twice or more, if the propagation contours (defined by the neighborhood used) do not coincide with the equidistance contours (defined by the metric chosen).

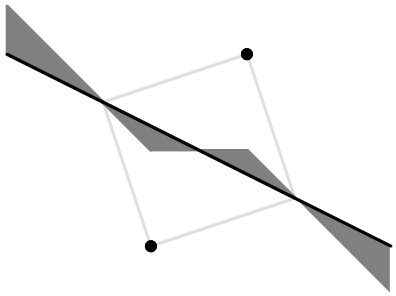


Figure 2. An image with two object pixels (black). The black line is the border between the two pixels' Voronoi polygons. Using quadratic propagation contours and Euclidean metric, the shaded areas will be updated twice, since the propagation from the more distant object pixel will arrive first.

The worst case for this problem is when a number of object pixels are located in a straight, slightly sloping line. In such a case, pixels in some areas may be updated once per object pixel in this sloping line, as illustrated in Figure 3.

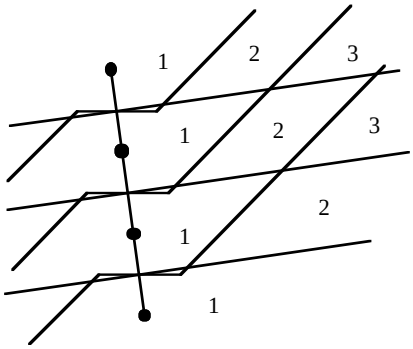


Figure 3. The worst case for the problem with multiple updates is a sloping line of object pixels. In the figure, only four object pixels are included, and in some areas, each pixel may get updated up to once per object pixel.

This problem can be avoided by accessing the pixels in order of increasing distance value. The optimal case is to make all updates of pixels in order of the distances that the pixels are assigned. We suggest two ways to achieve this, one exact and one approximating.

Verwer [11] suggest a method for integer-based non-Euclidean algorithms. Unnecessary updates are completely avoided at the cost of a fairly complicated algorithm, using *bucket sorting* of the Contour List. With slight modifications, the method can be adapted to Euclidean metrics, as is done in [6]. Montanari [3] uses a related algorithm for skeleton extraction, addressing the pixels in order of increasing distance.

A simpler method that still gives satisfactory results is to use *circular propagation by thresholding*. A threshold variable is introduced which holds an upper bound for the distance value of all pixels to be processed during the current iteration. Pixels with higher distance value remain in the contour set in the next iteration. After each iteration, this upper bound is increased by *the smallest distance from the mask center to its border*. This distance is 1 for a 3·3 mask, 2 for a 5·5 mask etc. For Chamfer 3-4, the distance is 3. For Euclidean DTs, it is 1. Experiments show that when using 3·3 neighborhoods, this does indeed remove almost all multiple updates, and that this has a significant impact on speed.

3. Neighborhoods

According to [5], theorem 3, the propagation of an EDT algorithm will work properly if the algorithm supports propagation of distance values along any straight line from any object pixel to infinity. It implies that we do *not* have to support propagation in other directions. Therefore we only need to check the pixels located beyond the processed pixel, looking from the object pixel.

In [3], theorem 1, Montanari shows that the *minimal path* between two points in an image is two straight lines when using metrics like City Block or Chamfer metrics. These two straight lines correspond to two of the many propagation directions supported by some neighborhood (mask). This means that to reach one point from another with the propagation of distance values in a DT, only two mask members need to be used. They are the two mask members supporting propagation in the directions closest to the direction from the origin pixel (object pixel) to the destination pixel, one on each side of the desired direction. This is illustrated in Figure 4.

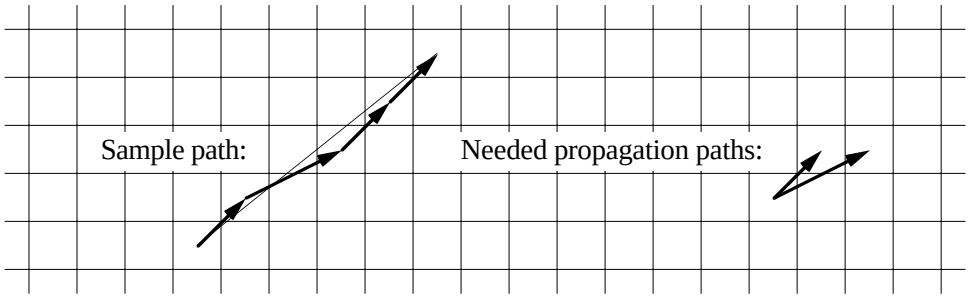


Figure 4. To propagate from one point to another, we need to use only the mask members with directions closest to the desired direction, one on each side.

Because of these two theorems, theorem 3 in [5] for Euclidean DTs and theorem 1 in [3] for non-Euclidean (pseudo-Euclidean) DTs, we may compute distance maps, Euclidean as well as non-Euclidean, using only two neighbors in most masks. Such masks, including neighbors only within a certain direction interval, is in the following called *directed masks*. This method was also used in [11], though not described in detail.

However, some reservations are needed to make the propagation complete, reaching to any pixel desired.

Each neighborhood member supports by itself propagation in a certain direction. We refer to these directions as the *directly supported* propagation directions (DSPD). In these directions, only one neighbor is needed. All other directions need to use a combination two DSPDs, as mentioned above.

Let us first consider 3·3 neighborhoods. There will be 8 DSPDs, which will divide the possible directions into 8 direction intervals. In each interval, different masks are used. These masks include only the center pixel and two other mask members.

If we assume that we have propagated a few steps out from the object pixels, the masks in Figure 5 should be used. Figure 6 illustrates the resulting propagation paths.

Note that the masks in the regions between two DSPDs are the union of the 2-pixel masks in these DSPDs.

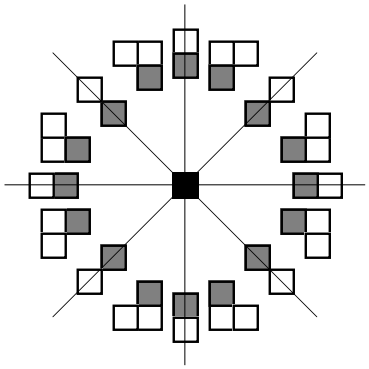


Figure 5. Directed masks. In directly supported propagation directions, only one neighbor need to be inspected, while otherwise two neighbors must be inspected.

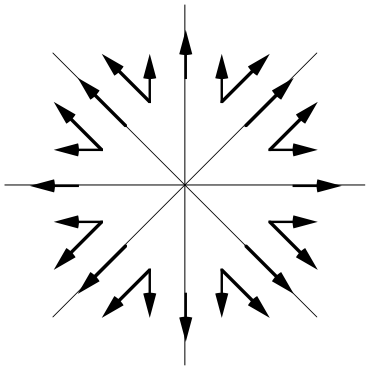


Figure 6. The resulting propagation paths in different directions in the previous figure.

The first two iterations need to be different. In the first iteration, we need to support a complete neighborhood, the 3·3 neighborhood shown in Figure 7.

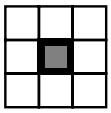


Figure 7. In the first iteration, propagation must be supported in all directions.

Using the neighborhood in Figure 7 in the first iteration, we need to use slightly modified masks in the second iteration as well. Otherwise, we would only propagate in the DSPDs. These masks are similar to the ones in Figure 5, but include more pixels in some cases. They are shown in Figure 8.

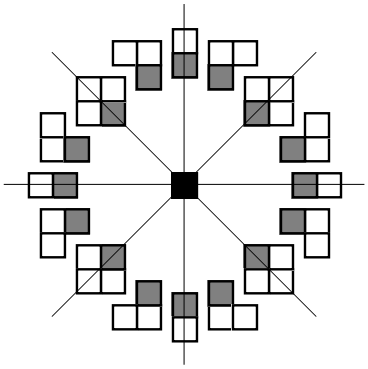


Figure 8. In the second iteration, the masks need to be slightly larger than in the following iterations.

Note that we only check one neighbor along coordinate axes and (after the second iteration) diagonals. If, for the horizontal case, we used a mask with, for example, three neighbors (up-right and down-right in Figure 9), we have a worst case for the algorithm in terms of computation time. Every time we have a straight horizontal or vertical edge in the image, large areas along these lines will be updated twice, as illustrated in Figure 10. However, this problem will not occur when using bucket sorting, but only when using circular propagation by thresholding or no circular propagation at all.

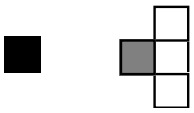


Figure 9. Possible, sub-optimal masks for use along horizontal or vertical lines from the object pixels. Apart from being larger, and thereby more time-consuming, we find a worst case that would make the algorithm even slower.

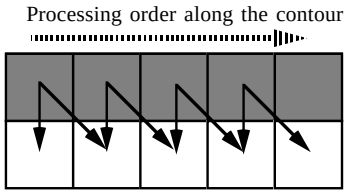


Figure 10. If we used the masks in the previous figure, all pixels in certain regions would be updated twice, once along a diagonal path and then along a horizontal or vertical path. The arrows show updates.

The following theorem justifies the correctness of these directed masks for EDT, especially the use of 1-neighbor masks under propagation horizontally, vertically and diagonally.

Theorem: In a Euclidean distance transform, propagation does not have to be supported from a diagonal vector under distance propagation to vectors with other directions except once, from the shortest possible diagonal vectors (±1,±1). We do not have to sup-

port propagation from a vertical or horizontal vector under distance propagation to vectors with other directions.

Proof: A proof is given in [7].[1]

An alternative to the modified second iteration is to use an even larger neighborhood in the first iteration, as shown in Figure 11. This simplifies the algorithms somewhat, but gives much unnecessary computation in cases where the binary image has many object pixels.

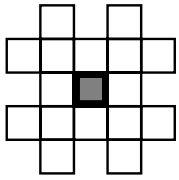


Figure 11. When processing the object pixels, the very first iteration, we need to check all eight neighbors, but we may consider an even larger neighborhood, including the pixels that are the closest ones that can not be reached along DSPDs.

Now, let us consider using these masks for EDTs. We do not only know the distance to the closest object pixel from each pixel, but also exactly where it is located. Hence, we have the necessary direction information for using directed masks. We also have no practical need for larger masks than 3·3, which simplifies the division in different direction intervals. This makes the directed masks very suitable for EDT. Hence, Figures 5-8 are immediately applicable.

For non-Euclidean distance, on the other hand, each pixel must hold not only the distance value, but also some directional information, like the index of a direction interval.

For example, consider the Chamfer-3-4 distance [1]. The mask is a 3·3 one, but depending on what direction interval the center pixel is in, only a few pixels are used in each case. The masks and propagation intervals are exactly as described above, as in Figures 6-8. The direction index can have 16 different values, one per interval and one per DSPD, as in Figure 12.

[1] The proof is also given in an appendix at the end of this chapter.

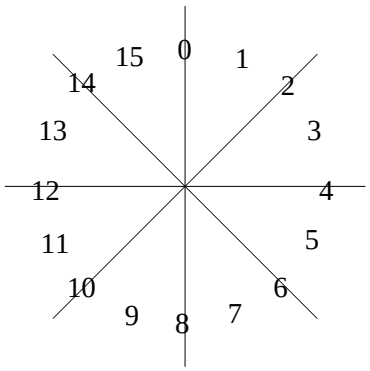


Figure 12. Possible indexes on different direction intervals for non-Euclidean transforms using a 3·3 mask.

As another example, consider a 5·5 neighborhood, like the Chamfer 5-7-11. We may make a similar division in intervals. We need twice the number of intervals, 16, but within each interval, we still need to propagate to no more than two neighbors. See Figures 13 and 14.

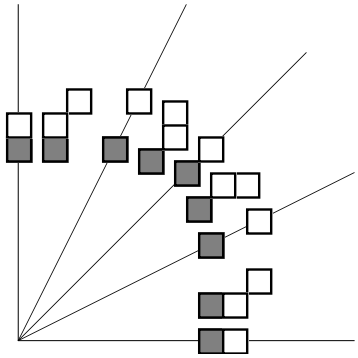


Figure 13. The masks in one quadrant for a non-Euclidean DT using a 5·5 neighborhood, such as Chamfer 5-7-11 or similar.

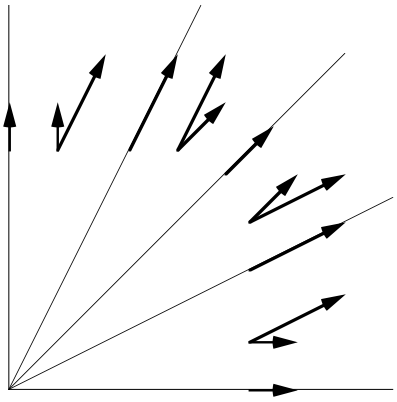


Figure 14. The propagation paths corresponding to the masks in the previous figure.

We also need modified masks for the first one or two iterations, as mentioned before (i.e. Figure 8). This method can be generalized for even larger neighborhoods, at the cost of

an even higher number of intervals. For these larger neighborhoods, circular propagation by tresholding, as described in the previous section, is no longer practical, but we should rather use bucket sorting.

4. Euclidean DT using Ordered Propagation

Below is pseudo code for Contour Processed Signed Euclidean Distance Transform (CSED), an algorithm generating Euclidean distance maps using ordered propagation with directed masks and circular propagation by thresholding, as described above.

In the following, D(v) is the length (distance) of the vector v. In order to make all computations with integer numbers, we will in a real program use the squared distance rather than the distance.

```
p is a pixel, defined by its position (px,py)
v(p) is the vector in p with components (vx,vy)
v(p)=(0,0) if p is an object pixel
v(p)=(N,N) if p is a background pixel, (N,N) is a vector long enough to be considered "infinite"
d is the distance threshold for circular propagation
o is an offset vector , the vector from the center pixel to a chosen neighbor

for all p
      if D(v(p))=0
            for all o ∈ {(0,1), (1,1), (1,0), (1,-1), (0,-1), (-1,-1), (-1,0), (-1,1)} test (o)
switch list 1 and 2
d := 1
† for all p in list1
      if D(v(p)) > d     p is put on list2
      else if vx(p)=0test (0,sgn(vy(p))   {vertically}
      else if vy(p)=0test (sgn(vx(p),0)   {horizontally}
      else if |vx(p)|=|vy(p)|test (sgn(vx(p)),sgn(vy(p)){diagonal}
                              if d = 1test (sgn(vx(p)),0), test (0,sgn(vy(p))
      else if |vx(p)|>|vy(p)|test (sgn(vx(p)),sgn(vy(p)), test (sgn(vx(p)),0)
      else                    test (sgn(vx(p)),sgn(vy(p)), test (0,sgn(vy(p)) {if |vx(p)|<|vy(p)|}
clear list 1
d := d+1
if list 2 is not empty, switch list 1 and 2 and repeat from †

subroutine test(o):
      if D(v(p+o)) > D(v(p)+o): v(p+o) := v(p)+o, p+o is put in list2
```

Note that the update rule is "D(...)>D(...)" and not "D(...)≥D(...)". If we allowed updates on equal distance, the speed would be reduced severely by redundant pointers in the Contour Set.

The pseudo code above is only a description of the fundamental algorithm. Many hardware-specific optimizations are possible, and should be considered when implementing the algorithm.

To test the speed improvement in practice, we made implementations in the C language and ran on a conventional workstation (Sun Sparcstation 2), measuring the processing time using the Unix command *time*. Two test images were used, one synthetic image with only a single object pixel in the center of the image, and one real image, shown in Figure 15. The size of both images were 256 by 256 pixels.

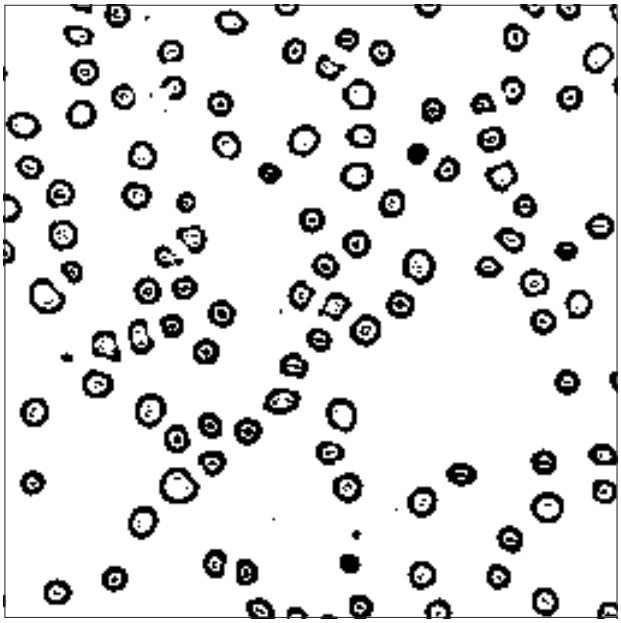


Figure 15. A test image: blood sample.

Four algorithms were tested. All were reasonably optimized, using lookup tables for distance value computations. The algorithms were 1) the scanning EDT (8SSED) [2, 10], 2) ordered propagation algorithm using complete 3·3 neighborhoods rather than the directed masks suggested in this paper, and the ordered propagation EDT (CSED) described above, 3) with and 4) without using circular propagation. The table below shows the processing times in seconds.

Algorithm	Image 1	Image 2 (Figure 15)
1. 8SSED	1.71	1.68
2. complete neighborhood EDT	1.29	1.68
3. CSED	0.77	0.81
4. CSED w/o circ. prop.	0.67	0.78

The ordered propagation algorithms would benefit even more than raster scanning algorithms from being implemented in assembly language, since these algorithms are more complicated. Despite the lack of low level optimization, CSED proved to be significantly faster than 8SSED, but at this level of optimization only when using directed masks. See also section 6 in this paper.

5. An error-free version of CSED

According to [2], raster scanning EDT algorithms are almost, but not totally error-free. The problem is that some Voronoi polygons (VP), for each object pixel the part of the image where it is the closest object pixel, have very sharp points, as exemplified by Figure 16.

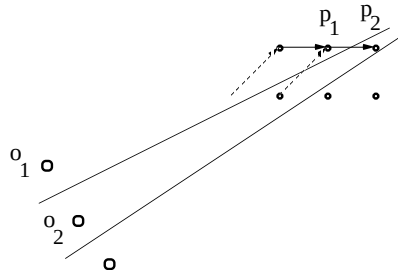


Figure 16. A VP can have very thin parts. When it fits between pixels in other VPs our neighborhood may be too small to reach some points. The arrows illustrate the queuing process in the image that will ensure error-free Euclidean distance maps.

In such cases, pixels in the thinnest part of the Voronoi polygon might not be within the neighborhood of other pixels in the Voronoi polygon, and thereby not directly reachable. The pixel may therefore not be reached by the propagation from the appropriate object pixel, and will receive an incorrect distance value (vector).

Ordered propagation algorithms should cause propagations similar to what a parallel algorithm would. Yamada shows in [8] that a parallel ($O(n^3)$) Euclidean distance transform algorithm (PED) like the one he proposes is totally error-free.

By a simple modification to the CSED, we may cause it to behave exactly as a PED, and therefore create error-free distance maps, but with complexity $O(n^2)$ instead of $O(n^3)$. One may get the impression that the CSED already behaves like a PED, but unfortunately, this is not true.

The reason for a PED to be error-free is that propagations from different object pixels may form queues in the image as illustrated in Figure 16. With chessboard propagation (quadratic) the propagation from the middle pixel (o_2 in Figure 16) will reach the narrow passage before the upper one (o_1), and may therefore use the pixel p_1 to reach the pixel p_2. The dotted arrows are propagation steps in one iteration, and the other arrows are propagation in the following iteration. The propagation from o_2 borrows p_1 for one iteration.

The parallel execution must work as implicitly assumed by Yamada, allowing p_1 and p_2 to be updated simultaneously, without one update affecting the other. This allows one propagation front to propagate immediately after another, which is what we need for Yamada's proof to hold.

The problem with the ordered propagation algorithms described above is that pixels that are to be processed in the iteration in progress sometimes are updated before they

are processed, thereby destroying the queuing process. If we can delay such updates, we will achieve the same result as parallel algorithms.

The solution is to save all updates and not execute them until the iteration is finished. The cost is that it takes even more memory than before, and we may not use circular propagation. We keep both the pointer to the pixel and the information to do the update in the lists. The update is made as a part of the next iteration.

Sometimes, several neighbors want to write in the same pixel during the same iteration. This means that we may not simply write all the information from the buffer to the distance map (of vectors), but we will have to check the present distance value each time.

Here follows pseudo code for the error-free CSED. It is similar to the pseudo code in the previous section. Note that the variable d is removed, and the new list member u is introduced, which is used for delaying the updates.

p is a pixel, defined by its position (p_x,p_y)
v(p) is the vector in p with components (v_x,v_y)
v(p)=(0,0) if p is an object pixel
v(p)=(N,N) if p is a background pixel, (N,N) is a vector long enough to be considered infinite
o is an offset vector, the vector from the center pixel to a chosen neighbor
u is a vector that v(p) will be updated to, if a better value hasn't shown up since the test that generated u

```
for all p
    if D(v(p))=0
        for all o ∈ {(0,1), (1,1), (1,0), (1,-1), (0,-1), (-1,-1), (-1,0), (-1,1)} test (o)
switch list 1 and 2
† for all p and u in list1
    if D(v(p)) > D(u)
        v(p) := u
        if vx(p)=0          test (0,sgn(vy(p)))
        else if vy(p)=0test (sgn(vx(p),0))
        else if |vx(p)|=|vy(p)|test (sgn(vx(p)),sgn(vy(p)))
                                if d = 1test (sgn(vx(p)),0), test (0,sgn(vy(p)))
        else if |vx(p)|>|vy(p)|test (sgn(vx(p)),sgn(vy(p))), test (sgn(vx(p)),0)
        else                    test (sgn(vx(p)),sgn(vy(p))), test (0,sgn(vy(p)))
clear list 1
if list 2 is not empty, switch list 1 and 2 and repeat from †

subroutine test(o):
    if D(v(p+o)) > D(v(p)+o): p+o and v(p)+o are put in list2
```

To verify the claims above in practice, experiments were made to confirm that the algorithm is indeed error-free, and to see how frequent the errors really are in the other algorithms. The test was made using images of 32·32 pixels. This may seem small, but the first errors for 8-neighbor algorithms (all but the 4SSED in the table below) were found at distance 13, and the largest known error [2] is at distance 18.4, well within the 32·32 image.

Three object pixels were put in the image in a number of possible arrangements. Pixel p_1 was placed along the bottom edge of the image, p_2 was placed along the left edge and p_3 within the triangle formed by p_1, p_2 and the lower left corner. The pixels were placed in all such combinations, and for every case, a number of distance transforms were run. This yields a $O(n^6)$ problem, generating 55970 different images. The test is not fully exhaustive with respect to recursive, scanning algorithms, but they were not the algorithms the test was focused upon.

The maximum error detected for each algorithm are also included in the table. These measures merely confirms the statements made in [2].

Algorithm	Max. error	Total # of errors
8SSED (scanning) [2,10]	0.074	158
4SSED (scanning) [2,10]	0.27	4221
Ordered propagation algorithm without circular propagation	0.054	27
Ordered propagation algorithm with circular propagation	0.074	130
Ordered propagation algorithm, error-free	0	0

The test has also been run using images of 64·64 pixels, but only for the error-free algorithm. We conclude that the test confirmed that the new algorithm was error-free, while all the others showed the expected errors.

6. Speed analysis

In this section, the speed of ordered propagation DTs is estimated and compared to corresponding sequential algorithms.

The measure chosen is *the number of memory accesses* necessary per pixel in the image. This is chosen since none of the algorithms in question demand any calculations other than additions, shifts and comparisons, and all algorithms have the same complexity, namely $O(N^2)$ for an $N \cdot N$ image.

Note that the correctness of the memory access measure is hardware dependent, and assumes that we may neglect both the instruction fetches and numerical calculations. That is, we assume that our hardware has good instruction caching and that calculations are fast, possibly pipelined. We also assume the bus to be wide enough to write a whole 2-component vector with only one memory access. Typically, this implies a 16-bit bus or wider.

To be able to make these measurements, we must outline how the calculations in the inner loops of the algorithms are computed. The operation of the DT algorithms are basically as follows:

Recursive, raster scanning algorithms:
for y, for x, scan the image:
Read the vector/distance value in the pixel x,y
Fetch the distance values/vectors of the neighbors
Compute the corresponding distance value for the center pixel for each neighbor
Compare
If a neighbor had a lower value, update the center pixel

Ordered propagation:
Fetch x,y from the list
Read the vector/distance value in the pixel x,y
Compute the corresponding distance value for each neighbor
Fetch the distance values/vectors of the neighbors
Compare
If a neighbor had a higher value, update it and write its position into the list

However, we need to ask ourselves how the distance value is calculated for Euclidean distances, where we only have the vectors.

We may simply compute x^2+y^2. It only requires multiplications, and is suitable if we can consider multiplications to be fast.

If we store the value x^2+y^2 for each pixel, we may compute new values incrementally, e.g. for the vector (x+1,y-1):

$$(x+1)^2+(y-1)^2 = x^2+2x+1+y^2-2y+1 = (x^2+y^2) + (2x-2y+2)$$

Thus, we only need additions and shifts.

We can also use a lookup-table, with a large two-dimensional array with either the distance $(x^2+y^2)^{1/2}$ or the square, x^2+y^2. If we have the distance value itself, we may also use it in the post-processing step to turn the vector image into a scalar distance map, if needed.

Then, how many memory accesses are needed for each kind of algorithm? We count the number for each case for a mask with M pixels including the center pixel:

Incremental sequential EDT (raster scanning):
M to read all distance values
M-1 to read all vector components except for the center pixel
2 to write if the pixel is changed

Look-up table supported sequential EDT (raster scanning):
M to read all vector components
M to read from the look-up table
1 to write if the pixel is changed

Non-Euclidean DT (raster scanning):
M to read all distance values
1 to write if the pixel is changed

Hence, we consider the inner loop to take the time 2M+1 for EDTs and M+1 for non-Euclidean DTs. Note that EDTs would only take M+1 if we consider multiplications negligeable, thereby removing the need for the look-up table.

Now, let us calculate the corresponding measures for some interesting ordered propagation algorithms:

Euclidean distance, ordered propagation, incremental implementation:
1 to read pointer to the center pixel
M to read distance values
1 to read the vector components for the center pixel
0 to (M-1) to write distance values (typically 1)
0 to (M-1) to write vector components (typically 1)
0 to (M-1) to write pointers (typically 1)

This sums to M+5. As noted above, M is 3 in most cases, but sometimes 2 or 4. Using M=3, this gives a total of 8.

Euclidean distance, ordered propagation, using look-up-tables:
1 to read pointer to the center pixel
M to read the vector components for all pixels

M for table look-up
0 to (M-1) to write vector components (typically 1)
0 to (M-1) to write pointers (typically 1)

M for table look-up is counted with incremental methods to modify the distance value of the center pixel. Using only table look-up, we would have to access the lookup table 2M-2 times. Totally, the sum is 2M+3, totalling to 9 for the 3·3 neighborhood case.

Non-Euclidean distance, ordered propagation:
1 to read pointer to the center pixel
M to read distance values
0 to (M-1) to write distance values (typically 1)
0 to (M-1) to write pointers (typically 1)

This sums to M+3.

If we use non-Euclidean algorithms with directed masks, as described in section 3.7, it would add one read and one write (typically), which gives a total of M+5.

With this information we may compare the speed of a number of different DTs. For sequential EDTs we only consider the 8-neighbor and 4-neighbor algorithms.

For the sequential algorithms, we get:

Raster scanning EDT:
4SSED [2]: $M_1 = 3$, $M_2 = 2$, total $(2M_1+1)\cdot 2 + (2M_2+1)\cdot 2 = 24$
8SSED [2]: $M_1 = 5$, $M_2 = 2$, total $(2M_1+1)\cdot 2 + (2M_2+1)\cdot 2 = 32$
4SSED/SIMD [5]: M = 3, total $(2M+1)\cdot 4 = 28$
8SSED/SIMD [5]: M = 4, total $(2M+1)\cdot 4 = 36$
8SSED/SIMD, 3 scans [5]: $M_1 = M_2 = 5$, $M_3 = 4$, total $(2M_1+1) + (2M_2+1) + (2M_3+1) = 31$

These values should be compared to corresponding values for non-Euclidean algorithms. Particularly, we should compare with the popular Chamfer DTs [1].

Raster scanning non-Euclidean DT:
City block distance: M = 3, total $(M+1)\cdot 2 = 8$
Chamfer 3·3: M = 5, total $(M+1)\cdot 2 = 12$
Chamfer 5·5: M = 9, total $(M+1)\cdot 2 = 20$
Chamfer 7·7: M = 17, total $(M+1)\cdot 2 = 36$

For the ordered propagation transforms, we may consider either using complete, symmetrical masks (M+3 for non-Euclidean metrics) or directed masks (M+5 for non-Euclidean metrics). First, we look at symmetrical masks:

Ordered propagation Non-Euclidean DTs with symmetric mask:
City Block distance: M=5, total M+3 = 8
Chamfer 3·3 or Chessboard Distance: M=9, total M+3 = 12
Chamfer 5·5: M=25, total M+3 = 28
Chamfer 7·7: M=49, total M+3 = 52

Non-Euclidean algorithms with directed masks take M+5 memory accesses, just like the Euclidean version. We therefore get the same values for both CSED and non-Euclidean algorithms:

Ordered propagation DTs with directed masks, Euclidean or non-Euclidean:
3·3 neighborhood: M≈3, total M+5 ≈ 8
5·5 neighborhood: M≈3, total M+5 ≈ 8
7·7 neighborhood: M≈3, total M+5 ≈ 8

This means that a Euclidean Ordered propagation DT is among the fastest DTs possible. Also, it is the most exact one of these fast algorithms. The non-Euclidean DTs are as fast, but not significantly faster than the CSED. We pay for this in two ways: we need more memory and a more complex algorithm.

For most general-purpose computer systems, the advantage is much less than indicated above. The numbers presented here describe the idealized case where only memory access limits speed. The fetching of instructions from memory is not included, and neither is the time for calculations. Even so, experiments have shown a significant speed improvement.

We conclude that the ordered propagation DTs are significantly faster than raster scanning distance transform algorithms, and that the difference in speed between Euclidean and non-Euclidean distance is likely to disappear completely with future improvements in processor architecture.

7. Conclusions

The method of using ordered propagation for computing Euclidean and non-Euclidean distance transforms was investigated. We have focused upon methods using a single list of pixel pointers as its representation of the propagation front.

We find that with ordered propagation algorithms, Euclidean distance maps are created faster than with other methods. The generation of non-Euclidean distance maps can also be speeded up by this method.

The possibilities for reducing the number of neighbors inspected during the processing of a pixel was investigated. The method chosen was directed masks, using masks including only pixels in the propagation direction. This was combined with circular propagation by tresholding, a simple method for accessing the pixels in approximately order of increasing distance value.

The typical neighborhood size is as low as 2 pixels, 3 including the center pixel. This is true even for high-precision Chamfer metrics, which otherwise demand very large neighborhoods.

An algorithm was found that generates error-free Euclidean distance maps with time complexity $O(N^2)$. This is by an order of magnitude faster than the previously known error-free parallel algorithms.

Finally, the algorithms using ordered propagation were compared to the raster scanning ones. A comparison counting the number of memory accesses needed indicated that the ordered propagation algorithms are significantly faster.

References

1. G. Borgefors, Distance Transformations in Digital Images, *Computer Vision, Graphics and Image Processing* **34**, 1986, pp. 344-371.
2. P.E. Danielsson, Euclidean Distance Mapping, *Computer Graphics and Image Processing* **14**, 1980, pp. 227-248.
3. U. Montanari, A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance, *Journal of the ACM* **15**, 1968, pp 600-624.
4. J. Piper, E. Granum, Computing Distance Transformations in Convex and Non-convex Domains, *Pattern Recognition* **20**, *1987*, pp 599-615.
5. I. Ragnemalm, The Euclidean Distance Transform and its implementation on SIMD architectures, *Proceedings, 6th Scandinavian Conf. on Image Analysis*, 1989, pp 379-384.
6. I. Ragnemalm, Fast erosion and dilation by contour processing and thresholding of Euclidean distance maps, *Pattern Recognition Letters* **13**, 1992, 161-166.
7. I. Ragnemalm, *Generation of Euclidean Distance Maps*, Licentiate thesis, no 206, Linköping University, Dept. of Electrical Engineering, Linköping, Sweden, Jan. 1990.
8. A. Rosenfeld, J.L. Pfaltz, Sequential Operations in Digital Picture Processing, *Journal of the ACM* **13**, 1966, pp 471-494.
9. H. Yamada, Complete Euclidean Distance Transformation By Parallel Operation, *Proceedings, 7:th International Conference on Pattern Recognition*, 1984, pp 69-71.
10. Q.Z. Ye, The Signed Euclidean Distance Transform and Its Applications, *Proceedings, 9:th International Conference on Pattern Recognition*, 1988, pp 495-499.
11. B.J.H. Verwer, Improved metrics in Image Processing applied to the Hidlitch Skeleton, *Proceedings, 9:th International Conf. on Pattern Recognition*, 1988, pp 137-142.
12. B.J.H. Verwer, P.W. Verbeek and S.T. Dekker, An Efficient Uniform Cost Algorithm Applied to Distance Transforms, *IEEE Trans. on Pattern Analysis and Machine Intelligence* **11**, 1989, pp 425-429.
13. L. Vincent, Exact Euclidean Distance Function by Chain Propagations, to be published.
14. L.J. van Vliet and B.J.H Verwer, A contour processing method for fast binary neighbourhood operations, *Pattern Recognition Letters* **7**, 1988, pp 27-36.
15. M. Fischler and P. Barrett, An Iconic Transform for Sketch Completion and Shape Abstraction, *Computer Graphics and Image Processing* **13**, 1980, pp. 334-360.
16. F. Leymarie and M.D. Levine, Fast Raster Scan Distance Propagation on the Discrete Rectangular Lattice, *CVGIP: Image Understanding* **55**, 1992, pp. 84-94.

Appendix: Proof of Theorem in section 3

The following proof was included in my licentiate thesis [7], but was removed before submitting this paper to CVGIP:IU, in order to get rid of some non-essential material and thereby having more interesting points per page. For completeness, while preserving the paper as close to the CVGIP:IU publication as possible, it is given here as an appendix. For historical reasons, the term “vicinity” is used below when referring to the Voronoi polygon of an object pixel. The Figure numbering is the one that was used in the licentiate thesis.

Theorem: In a Euclidean distance transform, propagation does not have to be supported from a diagonal vector under distance propagation to vectors with other directions except once, from the shortest possible diagonal vectors (±1,±1). We do not have to support propagation from a vertical or horizontal vector under distance propagation to vectors with other directions.

Proof: Consider the vicinity of every object pixel. If a pixel p_0 falls inside the vicinity of an object pixel f_0, in a distance map the distance value (vector) of p_0 should refer to f_0. Study Figure 3.22 with the pixel p_0, all pixels p_i that are not along any horizontal, vertical or diagonal line as shown in the figure and an object pixel f_0. The pixel p_0 is at one of the shortest possible diagonal vectors from f_0.

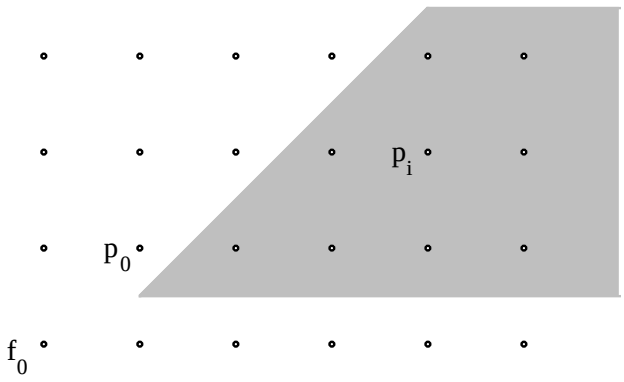


Figure 3.22. An object pixel f_0, a diagonal neighbor p_0 and an area with pixels p_i that are not located along any horizontal, vertical or diagonal line from f_0.

The theorem is valid if the following statement is correct:

If any pixel p_i within the area shown in Figure 3.22 is within the Voronoi polygon of f_0, p_0 is also within the Voronoi polygon.

If it is not correct, there exists another object pixel f_1 positioned so that p_0 falls within the vicinity of f_1, while some pixel p_i within the area falls within the vicinity of f_0. To find such positions, we draw a line through p_0. If this line is the possible border between the two Voronoi polygons, f_1 should be placed as a mirrored f_0 on the other side of the line. See Figure 3.23.

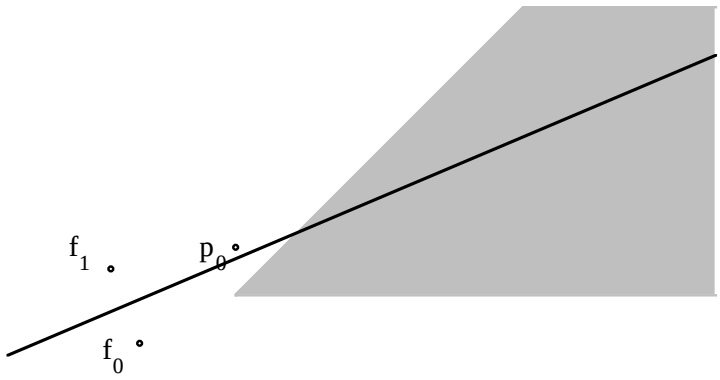


Figure 3.23. To find a counter-proof to the theorem, we must find an object pixel f_1 so that p_0 falls in the Voronoi polygon of f_1 and some pixel in the shaded area in the Voronoi polygon of f_0.

By simple geometry, we can find the area within which f_1 should be placed. First, we find that if the vicinity border passes through p_0, f_1 must be located at a circle with p_0 in the center and f_0 at the border, as illustrated in Figure 3.24.

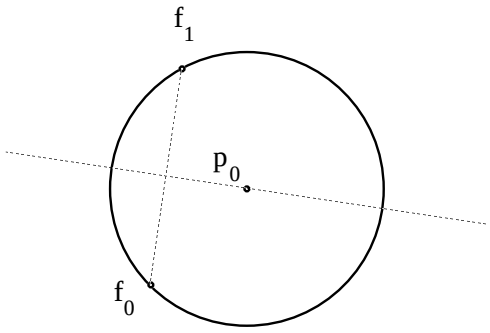


Figure 3.24. A point mirrored from f_0 over a line through p_0 must be on a circle with p_0 in the center and f_0 at the border.

We also know that, according to Figure 3.23, f_1 must be located to the left of f_0, otherwise the vicinity border would pass below the area we stated that it must cut through. This leaves us a small part of the circle, illustrated in Figure 3.25.

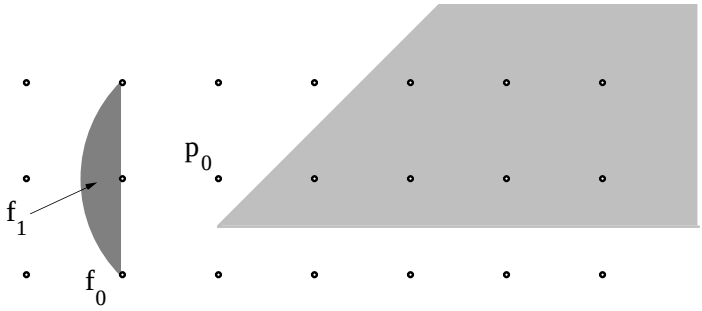


Figure 3.25. Can p_0 be closer to f_1 than f_0 while some shaded pixel is closer to f_0?

To make p_0 closer to f_1 that to f_0 while at least one pixel within the large shaded region is closer to f_0 than to f_1, the object pixel f_1 must be within the small shaded area to the left in Figure 3.25, and no grid pixel falls within this area.

Thus, it is impossible to exclude p_0 from the vicinity of f_0 without excluding the entire region, and thereby the theorem is proven.

The theorem says that if the propagation of distance values from f_0 should reach any pixel in the area shown, it should also reach p_0, and therefore we may safely use p_0 as a part of the path.

The Constrained Euclidean Distance Transform

Not yet published.
Short version in: Proceedings, Symposium on Image Analysis, Linköping 1990.

Det finns ingen kungsväg till geometri.

There is no royal road to geometry.

Euclids reply to King Ptolemy who asked if there was any shorter way to learn geometry than through the Elements.

The Constrained Euclidean Distance Transform

Ingemar Ragnemalm

Dept. of Electrical Engineering, University of Linköping, S-581 83 Linköping, Sweden

Abstract:
This paper presents an algorithm for generating constrained Euclidean distance maps. The algorithm can be used for finding the shortest path through an arbitrary 2-dimensional area with obstacles, sampled with rectangular grid. The known algorithms for Euclidean distance mapping can not easily be adapted for this problem. Hence, the algorithm presented in this paper differs considerably from other distance mapping algorithms.

1. Introduction

A *distance map* is an image where each pixel tells the distance to the closest set pixel (*feature* or *object* pixel) in a corresponding binary image. See Figure 1. Images and distance maps may have higher dimension than 2 [17], but in this paper, only 2-dimensional images will be considered.

	0	0	0	0	0	0	0		5	4	3	2	1	2	3
	0	0	0	0	1	0	0		4	3	2	1	0	1	2
	0	0	0	0	0	0	0		4	3	2	2	1	2	3
	0	0	0	0	0	0	0		3	2	1	2	2	3	4
	0	0	1	0	0	0	0		2	1	0	1	2	3	4
	0	0	0	0	0	0	0		3	2	1	2	3	4	5
a)	0	0	0	0	0	0	0	b)	4	3	2	3	4	5	6

Figure 1. From a binary image a) with two object pixels (1:s), a distance map b) can be generated. In the example, the City Block distance metric is used.

In ordinary distance maps, the distance values are just that, a single value telling the distance. An alternative is *Euclidean distance maps*, where each pixel carries a two-component vector that tells the distance to the closest object pixel along each coordinate axis [1,3,7,8]. For a vector (v_x,v_y) the distance is $(v_x^2 + v_y^2)^{1/2}$. As the name says, this distance is the exact distance. If this vector tells the exact position of the object pixel instead of just telling the distances, the distance map is said to be *signed*. Figure 2 shows a small signed Euclidean distance map.

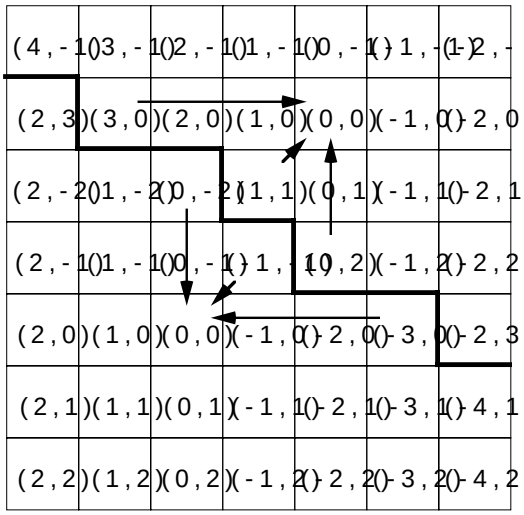


Figure 2. A signed Euclidean distance map generated by running the Signed Euclidean Distance Transform on the binary image in Figure 1a. A few arrows illustrate how the vectors point to the closest object pixel.

Note that algorithms that use only a single distance value per pixel and not a vector can never generate Euclidean distances if they are using neighbourhood operations (see be-

low). The distances generated are the distances along a path consisting of small steps between neighbour pixels (grid points). Such a path is straight only in special cases. This makes the errors increase with increasing distance. Apart from this, single component Euclidean distance values are often irrational and must be truncated to be used in a computer. Euclidean distance maps using vectors have none of these drawbacks. The fastest algorithms generate negligible errors in some cases, as noted by Danielsson [1]. These errors does not matter in practical cases, and if desired, error-free distance maps can be generated in slightly more time, as described in [8].

Some common metrics are listed in the table below:

Name	Definition
Euclidean distance	$L(x,y) = (x^2 + y^2)^{1/2}$
City Block distance	$L(x,y) = \|x\| + \|y\|$
Chessboard distance	$L(x,y) = \max(\|x\|, \|y\|)$
Octagonal distance	$L(x,y) = \max(\|x\|, \|y\|) + \min(\|x\|, \|y\|) \div 2$

The Euclidean distance metric can only be achieved with Euclidean distance maps (vector maps), generated by Euclidean distance transformations. Using approximative metrics like City Block, Chessboard or Octagonal distance, much simpler algorithms may be used [13]. Closer approximations to Euclidean distance has been suggested by Montanari [10] and Borgefors [14].

Constrained distance maps

A special case of distance maps is the constrained ones, where the source image consists not only of object pixels and background pixels, but also of obstacle pixels. A distance value in a constrained distance map tells the distance to the closest object pixel, but not along a straight line but along a path that doesn't pass through any obstacle pixels.

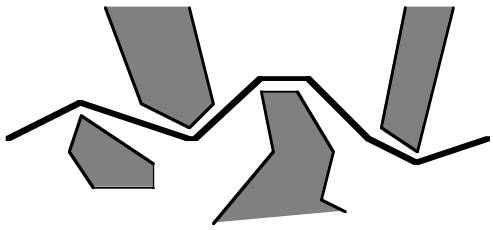


Figure 3. The typical output after using constrained distance maps: the shortest path from one point to another through an area with obstacles.

This kind of maps are typically used in path-planning problems. The basic method gives an approximative shortest path through a constrained area for a single-point object. In short, the algorithm works as follows:

1: Set the end point(s) to object pixels.
2: Generate the constrained distance map.
3: Find the path by beginning at the start point(s) and move to the lowest distance value in

the neighbourhood in each step. While following this path, it can be recorded in chain-code or similar form.

The method can be extended to find a path for rigid objects of arbitrary shape by expanding the obstacles (the constraining areas), using the object's shape as structure element. In short, every pixel where the object's center cannot be placed without it hitting an obstacle is set to an obstacle point. The algorithm can then be run as for a single-point object, as the algorithm above. This method is mentioned by Dorst et. al. as Fat Robots [6].

If the object is allowed to rotate, the distance map must be three-dimensional. The rotation angle spans the third axis. If it is also allowed to move in three dimensions, we must use a six-dimensional distance map, since rotation of a 3D object in a 3D space is defined by three angles. Still higher dimensions appear for problems with robot-arms, multiple objects etc.

The algorithms mentioned so far are all image processing algorithms, in the sense that they are pixel-based, with all data represented as sampled images. An alternative is to transform the obstacles into polygons and search for a path by examining the corners of the polygons. The output of such an algorithm is a shortest path computed in Euclidean metric, and should be equivalent to the output of the algorithm described below. See for example [5].

In some applications we may not necessarily want the shortest path, since we would be perfectly happy to get any path at all. [15] In other cases, we may want the safest path, where we can move some robot or other object with the largest possible clearance. In this paper, however, the problem being solved is to find the *shortest* path through an image with obstacles.

Generation of distance maps

The brute-force method for generating a distance map is to check each pixel against all object pixels. With an N·N image with P object pixels, this task has the complexity $O(P \cdot N^2) \approx O(N^4)$.

We define the following symbols:

I:	The set of all pixels in the image
F:	The set of object (or feature) pixels
B:	The set of background (or non-feature) pixels
O:	The set of obstacle pixels (exists in constrained distance maps)
D(i,j):	The distance between pixel i and j
D_i:	The distance value associated with pixel i

The brute-force algorithm can then be expressed like:

$$\forall\, i \in I: D_i = \min (D(i,j), j \in F)$$

Using neighbourhood operations, distance maps can be generated much faster.

In a neighbourhood operation, one pixel is the center of interest. We refer to it as the *center pixel*. Usually, its value (i.e. distance value) is determined as a function of itself and the members of a neighbourhood. This is called *read formalism* by Verwer et. al. [12]. Sometimes it is more suitable to use *write formalism*, where the value of each neighbour is determined as a function of the neighbour and the center pixel.

Despite its name, the neighbourhood does not necessarily include all pixels in a convex area around the center pixel, it is not necessarily symmetrical and the center pixel is not necessarily centered within the neighbourhood. The neighbourhood may be any suitable set of pixels, but their positions are always relative to the center pixel.

In a *parallel* algorithm for generation of distance maps, all pixels are processed in each iteration. We have to iterate until there are no changes in the distance map during an entire iteration. For generation of non-Euclidean distance maps, the algorithm can be expressed as follows:

N(i):the pixels in a chosen neighbourhood of i, excluding i itself
m:the iteration number

Initialization:

$D_i^0 := 0, i \in F$

$D_i^0 := \infty, i \in B$

$\forall\, i \in I: D_i^m := \min (D_i^{m-1}, D_j^{m-1} + D(i,j), j \in N(i))$

This is iterated until no change occur in the image. This procedure has a complexity of $O(N^3)$ instead of $O(N^4)$. Each pixel is tested against neighbours in all directions, so distance values will propagate in all directions, one layer per iteration, as illustrated in Figure 4.

The Euclidean versions are slightly different, since they don't keep single distance values but rather vectors.

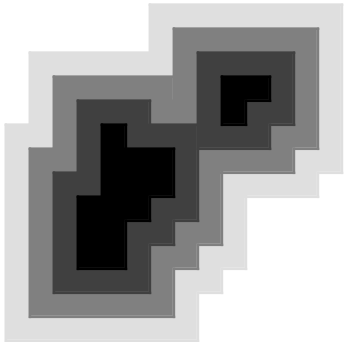


Figure 4. With a parallel algorithm, each iteration will cause one step of propagation. In the figure, the objects are black and each shade of grey corresponds to one iteration.

For single processor computers, a faster method for generate distance maps of either kind is to use a *recursive* algorithm that scans the image twice, once in each direction. When a pixel is processed (referred to as the *center pixel*), its distance value is compared to the distance value of a number of neighbours (modified to refer to the center pixel) and the lowest value found is written into the center pixel. This will cause the distance values (or vectors) to propagate from each object pixel in the direction of the scan. Recursive algorithms have the complexity $O(N^2)$. See Figure 5. The Figure also includes two masks, illustrating what neighbours are used in each scan. See further below.

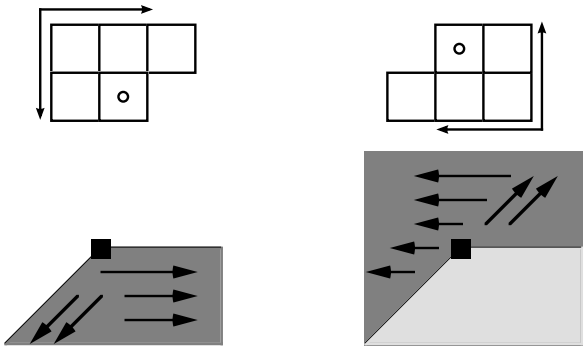


Figure 5. Using a recursive algorithm, the first scan will cause propagation over a part of the image and (the) following scan(s) will cover the rest of the image.

The algorithm can be expressed as follows:

(x,y): the pixel at the coordinates x,y.
$N_f(x,y)$:the pixels within the neighbourhood of (x,y) chosen for the forward scan, excluding the center pixel.
$N_b(x,y)$:the pixels within the neighbourhood of (x,y) chosen for the backward scan, excluding the center pixel.

Initialization:

$D_i^0 := 0, i \in F$

$D_i^0 := \infty, i \in B$

Forward scan:

for x:=1 to N
    for y:=1 to N
        $D_{x,y} := \min \{D_{x,y}, D_{z,w} + D((x,y), (z,w))\}, (z,w) \in N_f(x,y)$

Backward scan:

for x:=N to 1
    for y:=N to 1
        $D_{x,y} := \min \{D_{x,y}, D_{z,w} + D((x,y), (z,w))\}, (z,w) \in N_b(x,y)$

Thus the distance value $D_{x,y}$ may be changed to $D_{z,w} + d$, where $d = D((x,y), (z,w))$ which is the distance from (x,y) to (z,w). Figure 6 shows neighbourhoods (masks) N_f and N_b

including distances within the neighbourhoods for the Chamfer 3-4 metric as suggested by Borgefors [14].

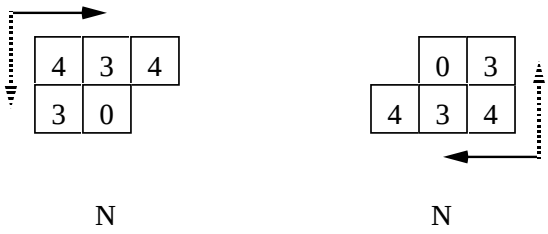


Figure 6. Neighbourhoods (masks) for the Chamfer 3-4 metric.

The Euclidean version is slightly different. With Euclidean metric, we can no longer use only two scans, but must use three or four [7] or two "double" scans [1]. Below is a general algorithm that performs one of these scans. By determining a number of neighbourhoods and corresponding scanning patterns, we can define the complete algorithm.

$N_p(x,y)$:the pixels within the neighbourhood of (x,y) chosen for the scan p.
$\bar{v}_{x,y:p}$the vector of the pixel at (x,y) after iteration p.

Initialization:
$\bar{v}_{i:0} := (0,0),\ i \in F$
$\bar{v}_{i:0} := (\infty,\infty),\ i \in B$

Scan number p:
for x or y
  for y or x
    $\bar{v}_{x,y:p} := \min \{\bar{v}_{x,y:p-1}, \bar{v}_{z,w:p} + (x-z,y-w)\},\ (z,w) \in N_p(x,y)$

The "min" function above returns the shortest vector.

In the following figures, two recursive Euclidean distance transforms are defined by their masks.

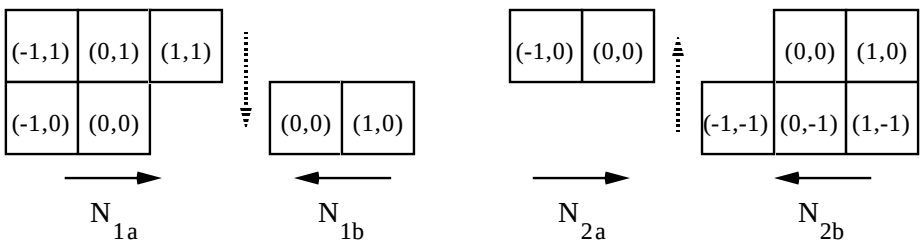


Figure 7. The neighbourhoods (masks) for the 8SSED algorithm for generation of signed Euclidean distance maps [1]. The scans N_{1a} and N_{1b} are performed simultaneously, applying both for each row of the image. The same holds for the scans N_{2a} and N_{2b}.

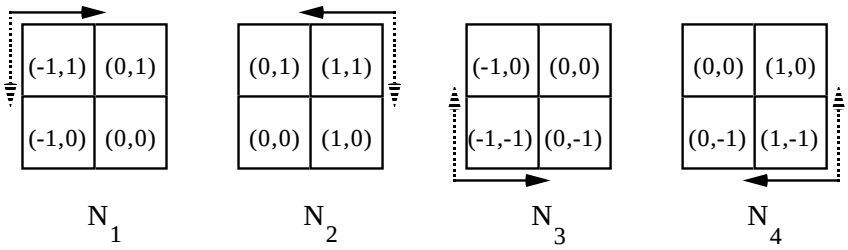


Figure 8. The neighbourhoods (masks) for the 8SSED/SIMD algorithm for generation of Euclidean distance maps [7].

The third alternative is to use *ordered propagation* (or *contour processing) methods*. These methods are similar to the parallel ones, but keep trace of the propagation front, and only test pixels in that part of the image [8, 9, 11]. Like the recursive algorithms, they have the complexity $O(N^2)$, but they are faster by a constant factor since each pixel is visited only once instead of two (non-Euclidean metric), three or four (Euclidean metric).

The contour processing algorithms in [8] can be expressed in pseudo code as follows:

C_1, C_2:the set of pixels in the active contour, the propagation front, henceforth referred to as the Contour Set. C_1 is the active contour in the current iteration, while C_2 is built to become the set used as C_1 in the next iteration.

```
Initialization:
D_i^0 := 0, i∈F
D_i^0 := ∞, i∈B

C1 := F
C2 := ∅
†
∀ i∈C1:
      ∀j ∈ N(i):
            if Dj > Di+D(i,j)
                  Dj := Di+D(i,j)
                  C2 := C2 ∪ j
if C2 is not empty:
      C1 := C2
      C2 := ∅
      Repeat from †
```

This is expressed using non-Euclidean metric, but the algorithm is easily modified for Euclidean distance, using vectors. With vectors, the neighbourhoods N(i) can be modified to include only pixels on the far side of the center pixel viewed from the object pixel, which makes the algorithm even faster.

As mentioned above, the Contour Set is the set of pixels at the propagation front. This set is represented by a list or queue which will be referred to as the Contour List.

Generation of constrained distance maps

Constrained distance maps using non-Euclidean distance can be generated using similar methods as for the non-constrained ones. In this case, however, contour processing algorithms have a major speed advantage. For recursive (scanning) algorithms [6] and parallel algorithms, severe worst cases exist, suggested by Danielsson et. al. [16]. Contour processing algorithms have no such worst case for constrained distance maps [8, 12].

A parallel algorithm for constrained distance maps propagates of one step per iteration. Such an algorithm is similar to its non-constrained equivalent, the only difference being that each step should check for obstacle pixels.

See Figure 9, which shows a simple image with obstacles and a single object pixel.

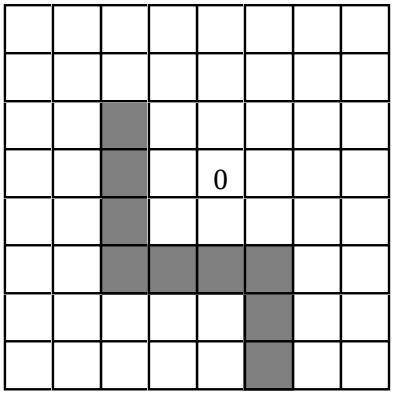


Figure 9. A simple image with obstacles (grey) and a single object pixel (0).

If we use the simple Chessboard distance metric, a 3·3 neighbourhood and a parallel algorithm, the algorithm will give intermediate results as shown in the figures 10 and 11, below.

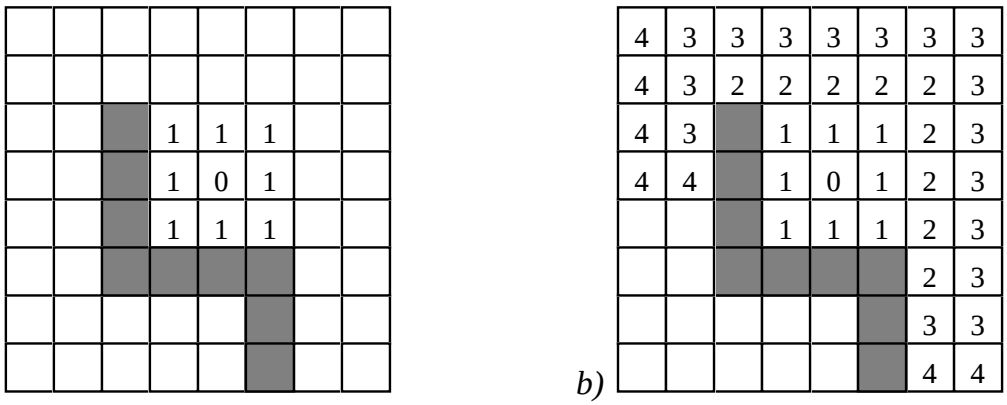


Figure 10. The distance map generated by a parallel algorithm after a) 1 iteration and b) four iterations.

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

Figure 11. The resulting constrained distance map.

A recursive algorithm uses forward and backward scans like the non-constrained versions, but they are repeated until there is no changes in the distance map during an entire iteration. A recursive algorithm is described in [6]. Figure 12 illustrates a recursive algorithm using the Chessboard metric.

Piper and Granum [9] suggest computing constrained distance maps recursively with more than two masks. They show that this is needed for some convex domains.

Contour processing methods will propagate exactly as a parallel algorithm, but in each iteration, only the pixels at the propagation border will be processed. [8,9,12]

In the general case, we have an image with obstacle pixels, background pixels and a number of object pixels. The constrained distance transform propagates from the object pixels to all accessible parts of the image. Thus, after termination of the algorithm, all accessible background pixels will hold the distance value to the closest object pixel. Now, consider the two background pixels for which

1) the path to the closest object pixel is longest, and

2) the path to the closest object pixels make the largest number of turns.

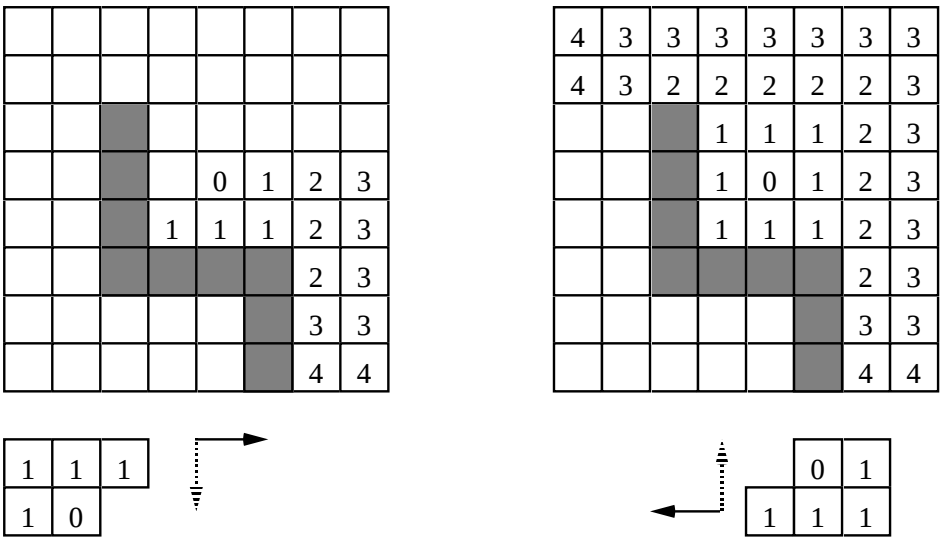


Figure 12. The same constrained distance map being computed by a recursive algorithm. After two scans, the distance map is still not complete, and more iterations are needed.

The computation time of a parallel algorithm will depend on the longest path that has to be generated, the path to pixel 1) above. One iteration per unit step along this path is needed. Thus, the worst case is an image with the longest possible path.

The computation of a recursive, raster scanning algorithm will depend on the path with the largest number of turns, the path to pixel 2) above. The worst case is an image where propagation must be performed along a long path with many turns, like the Meander curve shown in Figure 13. For constrained contour processing distance transforms, this case poses no speed problem.

The computation of a contour processed algorithm will depend on the number of background pixels in the image. If this number can be considered proportional to the image size, which is true in most cases, we may consider the execution time of the algorithm data independent.

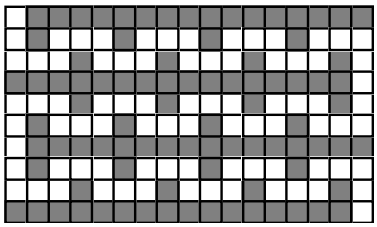


Figure 13. A Meander curve, the worst case for constrained distance transform based on the conventional recursive algorithms.

The algorithms described above use non-Euclidean metrics, with only a single distance value per pixel. By using the improved metrics suggested by Borgefors [14] we can get arbitrarily good approximations to Euclidean distance. However, good approximations imply using larger neighbourhoods that 3·3 pixels. For large neighbourhoods the propa-

gation of distance values can pass through thin obstacles. Using a 5·5 neighbourhood, 1 pixel wide obstacles would be ignored, 7·7 would pass through 2 pixel wide obstacles and so on. See Figure 14.

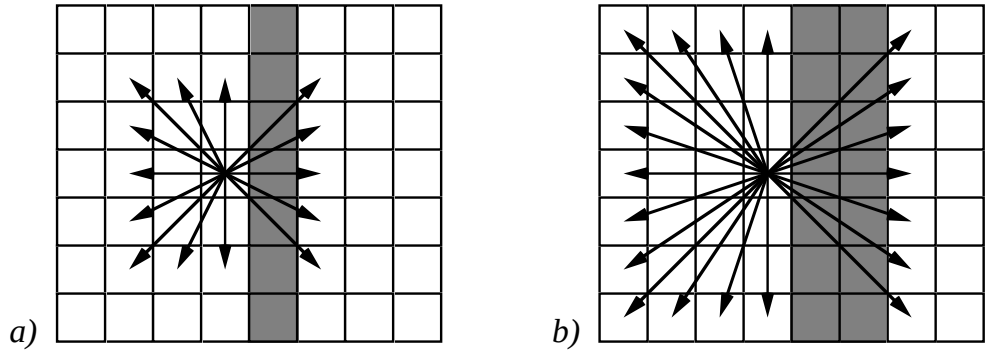


Figure 14. If a 5·5 neighbourhood is used for generating constrained distance maps, the propagation of distance values will be able to pass through 1 pixel wide obstacles (a). If a 7·7 neighbourhood is used, the propagation of distance values will be able to pass through 2 pixel wide obstacles (b).

Dorst and Verbeek [6] suggests both a 3·3 neighbourhood and a 5·5 one. In the 5·5 case, they note that obstacles need to be at least 2 pixels thick to avoid “leaking”, which seems to be a rather unfortunate limitation.

A circumvention of this problem not mentioned in [6] is to check the farthest neighbours only if the closer neighbours were not obstacles. The exact scheme for this solution depends on the connectivity definition used for the obstacles. For each neighbour outside the 3·3 neighbourhood, we must test a number of pixels between the neighbour and the center pixel for obstacle pixels.

If we want to maintain a reasonable definition of an obstacle and avoid the cumbersome testing for obstacle pixels between the pixel being updated and the center pixel, as mentioned above, we must use a 3·3 neighbourhood (or smaller). Thus, the best approximation of the Euclidean metric that can be obtained by the present methods without demanding some lower limit on obstacle width is the 3·3 neighbourhoods suggested by Borgefors [14], with the Chamfer 3-4 distance as a good integer approximation. One such algorithm is the 3·3 neighbourhood case in [6].

The Euclidean, vector-based distance transform produces totally or almost error-free distance maps using only 3·3 neighbourhoods. This property should make such vector-based techniques very suitable for constrained distance mapping.

However, it is not trivial to adapt Euclidean distance to constrained distance maps, since the vectors used in an Euclidean distance map are totally independent of the path. (See chapter 4.) This problem, generating path dependent Euclidean distance maps, is what the rest of this paper is all about.

To illustrate why the conventional Euclidean distance mapping can not be used directly for constrained distance maps, we use the example from Figure 9-12, see Figure 15. With a parallel or ordered propagation (contour processed) algorithm, the first two iterations give the intermediate results shown below.

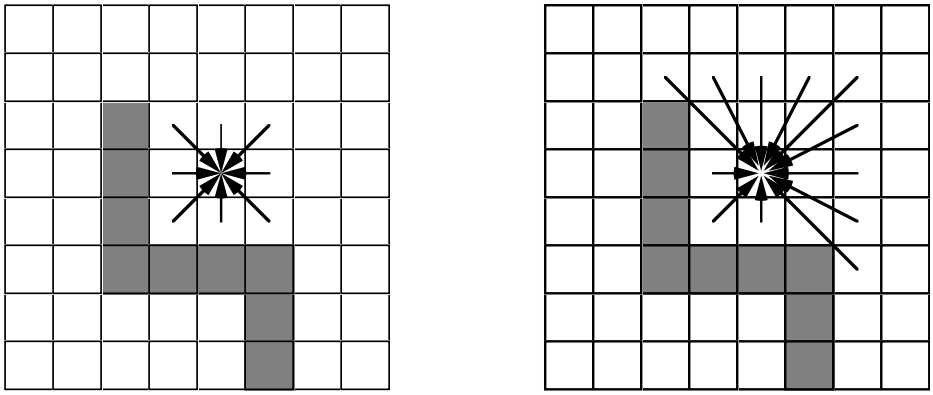


Figure 15. The intermediate results from the two first iterations of a conventional Euclidean distance transform in a constrained image.

As long as the propagation takes no turns, everything is good, but when we do (in the third iteration in the example) we get false and/or meaningless results. The vectors generated cross over obstacles.

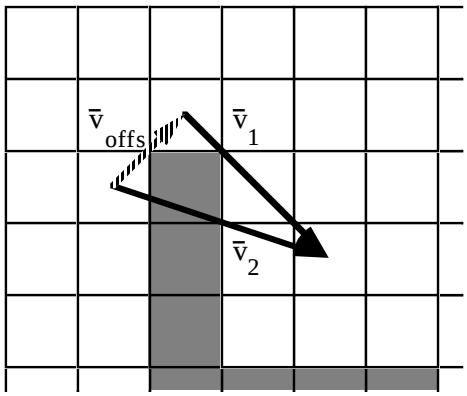


Figure 16. A part of the previous figure. In the third iteration, a new vector $\bar{v}_2$ is generated as the sum of the vector in a neighbour, $\bar{v}_1$, and the path between the two pixels, $\bar{v}_{offs}$. The resulting vector passes straight through obstacles.

Thus, when the propagation front propagates around a corner, some checking must be made to detect it and the turn must be remembered in some way. In the example, the two vectors should be kept as parts of the resulting path rather than being added together. Obviously, we need a data structure that can represent a turning path.

2. Vector chains

The ordinary, non-constrained Euclidean distance transform is based on vectors instead of distance values, where the vectors point to the closest object pixel. The corresponding case for a constrained distance map is a *chain of vectors*, where each vector points to the pixel "in sight" to which the resulting vector chain is shortest. The last pixel in the vector chain holds a vector that points to the pixel itself (the vector (0,0)).

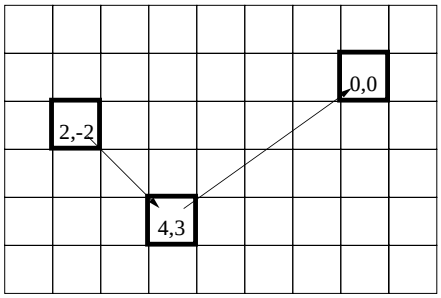


Figure 17. A short vector chain, with two links (vectors). The first link points from the leftmost of the marked pixels to the middle one. The chain ends with the rightmost of the marked pixels, the vector of which points to the pixel itself.

A constrained Euclidean distance map, a vector map with vector chains, will have the following properties:

Definition: All pixels $i \in I$ hold a vector $\bar{v}_i=(v_{xi}, v_{yi})$.

Definition: The vectors $\bar{v}_i$ are relative to the pixel they are stored in. The coordinates of the pixel i is denoted by $\bar{i}$. The next pixel in the chain from the pixel i can be found by adding the vector $\bar{v}_i$ to the coordinates of i, that is, $\bar{i}+\bar{v}_i$. There must be a straight path without obstacles from i to $\bar{i}+\bar{v}_i$ for the vector $\bar{v}_i$ to be valid.

Definition: All object pixels, $f \in F$, hold the vector $\bar{v}_f=(0,0)$. Thus, in every object pixel, the vector points to the object pixel itself.

In the following, the word "link" will occasionally be used for a vector in a vector chain.

Since the vector chains are generated by a propagation process, starting at the object pixels, we will refer to the pixel pointed to by the vector in the pixel i as the *originating pixel* of the vector in i.

If we want to find the shortest path from an arbitrary non-obstacle pixel $i \notin O$ to the closest object pixel, we may find it by the following recursion which also defines a chain of vectors.

$$\mathrm{Path}(i) = \begin{cases} \bar{v}_i = (0,0): \varnothing \\ \bar{v}_i \neq (0,0): (\bar{v}_i,\ \mathrm{Path}(\bar{i} + \bar{v}_i)) \end{cases}$$

A vector chain holds the complete information about a path. The path can thus be extracted from the image and stored as a small data structure. Normally, it is far more economic than for example chain-code, since it may contain long vectors that pass over large open areas with a single step. The two component vector is also an exact representation of the true distance, though the endpoints of each vector must naturally be in the sampling grid.

3. Restricting propagation to line-of-sight

According to the definition of vector chains in the previous section, a vector must not point to a pixel outside its “field of view”. This means that there has to be a straight path without obstacles between the originating pixel and the end of the vector.

As mentioned above, an ordinary Euclidean distance transform delivers a result in each pixel which is a single vector pointing directly to the closest object pixel, as a straight path regardless of all obstacles.

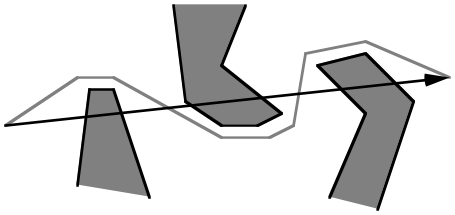


Figure 18. If propagation of vectors in an Euclidean distance transform is unrestricted, possible in all directions from any position even from non-object pixels, it will pass around any obstacle (which is desirable) and result in a single vector that cuts straight through any obstacles (which makes the result useless).

To cope with the constrained space, we have to restrict the propagation in some way, and make a new link for each turn. To begin with, however, let us forget about the turns. The first problem is to generate an Euclidean distance transform that covers only the areas directly visible from the object pixels. See figure 19. When this problem is solved, we will return to the problem of making turns.

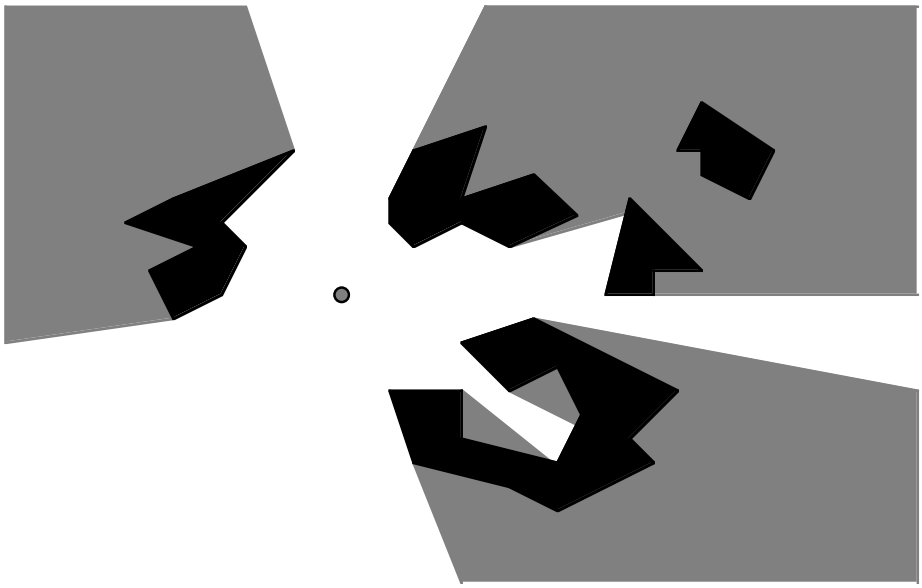


Figure 19. If propagation is restricted to line-of sight, the propagation from the object pixel in the figure (small shaded circle) should not reach the shaded areas since these areas are occluded by the obstacles (black).

In the following, the term *object pixel* will generally be replaced by *originating pixel* (as mentioned before) since the results will be used for generating vector chains. Then, a vector will not always point to an object pixel but rather to the next link in the vector chain.

The line-of-sight problem can be solved by keeping information about the path used by the propagation. In principle, propagation is only allowed *along straight lines*, like beams of light from the originating pixels. We need some mechanism for restricting the propagation accordingly. Since we are working in a sampled image, a grid, arbitrary straight lines will not hit pixels exactly. Therefore, the idea about propagation along straight lines must be modified to fit the grid.

Given a pixel, its vector, its originating pixel and the path used to propagate to it from the originating pixel, we can find an area that can be considered “forward”. Propagation is allowed from the pixel only to pixels within the “forward” area. The forward area is defined by the *(allowed) Direction Interval*, DI.

Similarly, given only a pixel, its vector and its originating pixel, we can find its *Locally defined allowed Direction Interval*, LDI, which is an interval including all directions that could be allowed with any path from the originating pixel to the vector. The allowed direction interval (DI) of the propagation from the pixel is then the intersection of all LDIs in the path from the originating pixel to the pixel.

We will need the following definitions:

Definition: A single obstacle pixel is square-shaped, side length one pixel distance and centered around the point specified by the pixel's coordinates.

Definition: The obstacles are 8-connected. A path may not pass between two connected obstacle pixels.

Definition: The background is 4-connected. This follows from the previous definition.

Definition: The object that is to be moved through the image is square-shaped, side length one pixel distance and centered around the point specified by the object's coordinates.

Note that the definitions of the obstacle pixels and the moving object is equivalent to a point-shaped moving object and obstacles with the side length *two* pixel distances, since the centres of the moving object and the obstacle pixels must be one pixel distance apart in either case. See Figure 20.

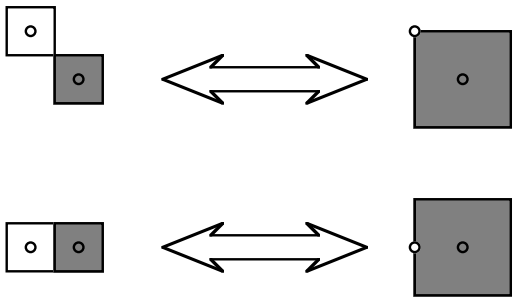


Figure 20. Using 1·1 pixel distance moving object (white) and 1·1 pixel distance obstacle pixels is equivalent to point-shaped moving object and 2·2 pixel distance obstacle pixels with respect to the allowed positions of the center of the moving object.

In practice, the object that is supposed to travel along the path is not always a single pixel. This is handled by expanding the background appropriately, using the object as structural element or expanding to its maximum radius.

We will now introduce the unit vectors $\hat{v}_{major}$ and $\hat{v}_{minor}$, which are important for the algorithm. These unit vectors indicate the orientation and sign of the dominant and the non-dominant component of a vector, respectively, as illustrated in Figure 21.

The representation of the propagation front: pixel blocks

During the generation of the constrained distance map, we must keep track of the allowed direction interval of every propagation front. Considering that we must have this extra information, it follows that we must use an ordered propagation algorithm similar to the contour processed Euclidean distance transform [8].

In all the distance mapping algorithms described in the introduction, the processing is performed *pixelwise*, that is, with a single center pixel at a time. This single center pixel is compared to the pixels in a neighbourhood. For a pixelwise ordered propagation algorithm, the propagation front is represented as a set of pixels, above referred to as the Contour Set. The Contour Set is considered identical to the list of pointers defining the contour, the Contour List.

Unfortunately, this approach is *not* suitable for vector-based constrained distance mapping. In this section, we will briefly describe how it would work, and why it fails. In spite of its failure, the approach illustrates mechanisms used in the final solution.

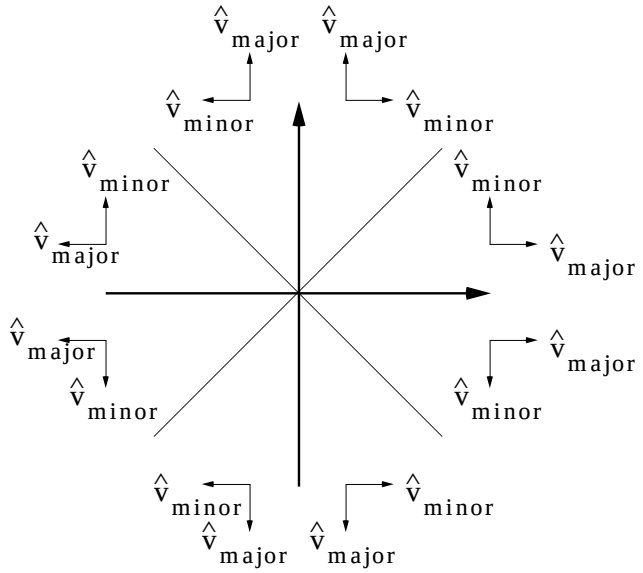


Figure 21. The vectors shown above are used to simplify and generalize the descriptions below.

For a pixel located at $\bar{c} = (x,y)$ with a vector $\bar{v}$ that is not vertical, horizontal or diagonal, the LDI as defined above must be within the lines through the two neighbour pixels situated to the left and to the right of $\bar{c}$, viewed from the originating pixel $\bar{c} + \bar{v}$, the *one-pixel-out* neighbours. See Figure 22. If the vector is expressed the usual way in two components as $\bar{v} = x \cdot \hat{x} + y \cdot \hat{y} = \bar{v} \cdot \hat{v}_{major} + \bar{v} \cdot \hat{v}_{minor}$, these pixels are the pixels closest to $\bar{c}$ along the $\hat{v}_{minor}$ axis, $\bar{c} - \hat{v}_{minor}$ and $\bar{c} + \hat{v}_{minor}$.

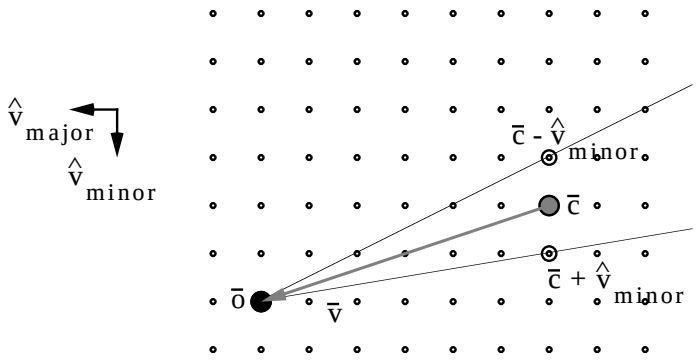


Figure 22. The LDI of a pixel $\bar{c}$ when using an 8-pixel neighbourhood. The propagation from a pixel $\bar{c}$ which causes vectors to the same originating pixel $\bar{o}$ must be restricted to a region bounded by the lines passing through both $\bar{o}$ and either of the two neighbours along the non-dominant component of the vector $\hat{v}_{minor}$ (vertical in the figure).

This is the *one-pixel-out rule*, which holds for the d^8 case as written above for all vectors except diagonal ones, where it has to be slightly modified. For other neighbourhoods (i.e. d^4) similar rules can be made. The idea is that if a (pixelwise) propagation that passes $\bar{c}$ instead of one of the one-pixel-out neighbours, the following propagation steps should never go beyond the line passing through the one-pixel-out neighbours, that is, outside

the LDI, since it then would not follow a straight line any more. Propagation outside the LDI should be handled by propagation through other pixels, not through $\bar{c}$.

If there are obstacles in the neighbourhood of $\bar{c}$, the LDI should also be influenced by them. If we keep the allowed direction interval (DI) that is valid in each propagation step, the DI of each step can be found as the intersection of the DI in the previous step and the LDI. This is illustrated in Figure 23 for a certain path without any obstacles. Each step is numbered, and the bounds are shown and marked with the corresponding number.

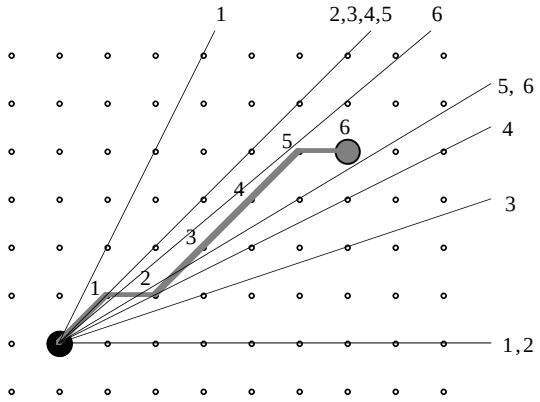


Figure 23. An example of how the DI is updated.

This updating is needed to ensure that propagation indeed is limited to straight lines, and not able to take turns around obstacles. We need to accept that pointers to one and the same pixel may occur more than once in the Contour List, as exemplified in Figure 24, below. If we updated a pixel, and included it in the Contour List, only if its distance value was *less* than the old one in the pixel tested, we would cut off some propagation paths. If, for example, the black propagation path reaches the pixel before the dotted one, propagation will not be possible in the area between the two lower bounds in the figure. Thus, we must update on *less or equal* distance.

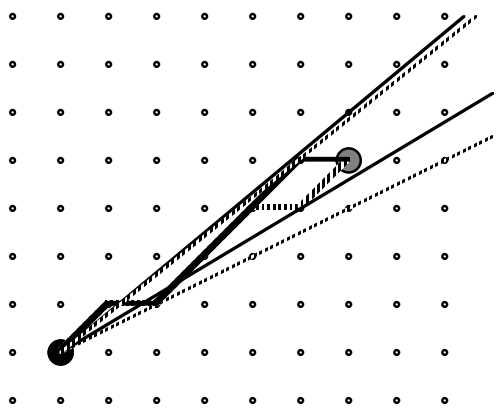


Figure 24. Sometimes, we need to have two pointers to the same object pixel at the same time, since the direction interval is depending on the path. In the figure, the bounds for the two paths are drawn with corresponding patterns (black and dotted).

Unfortunately, this means that we can get any number of pointers to one pixel in the Contour List. Practical experiments indicate that the numbers of pointers to each pixel on the propagation contour grows approximately as a linear function of the distance from the originating pixel. This makes the algorithm very slow, especially for images with large open spaces. Hence, we must use another representation of the propagation front.

In the final constrained Euclidean distance transformation algorithm, pixelwise propagation is abandoned, and instead we use *pixel blocks*. Each entry in the Contour List refers to a number of pixels, located in a straight line along $\hat{\mathbf{v}}_{\text{minor}}$. Such a pixel block represents an interval of allowed propagation directions. There is one direction interval stored per block instead of one per pixel.

See Figure 25 for an example. The interval shown in the figure is the largest possible for the given pixel block. The direction interval of the pixel block is the union of the direction intervals of all the pixels in the pixel block.

A pixel block must work within one octant only, so that $\hat{\mathbf{v}}_{\text{minor}}$ and $\hat{\mathbf{v}}_{\text{major}}$ will be uniquely defined for the whole block. Thus, the end points of the pixel block may be located within the octant or on the edges of the octant.

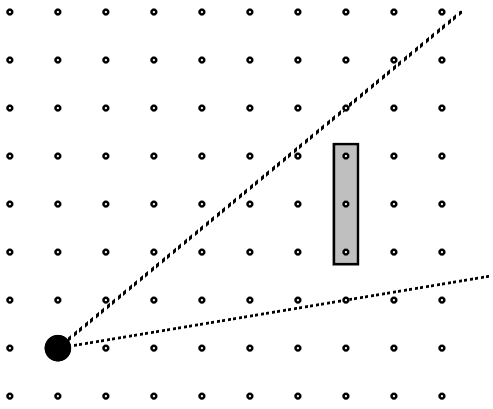


Figure 25. Instead of processing one pixel at a time, we use "pixel blocks".

When a pixel block is processed, the result is a number of pixel blocks, typically one, possibly several or none, depending on obstacles and other propagation fronts. They are located one step beyond the pixel block that was processed, as illustrated in Figure 26.

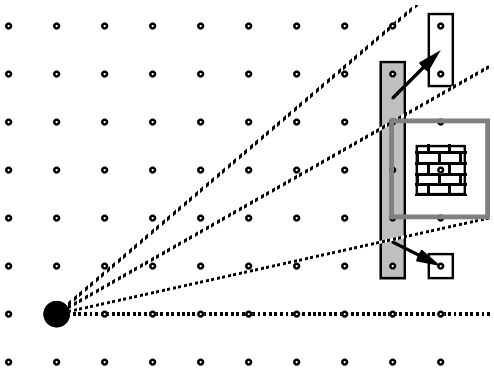


Figure 26. When a pixel block (grey) is processed, a number of pixel blocks one step ahead are produced. In the figure, the result is two blocks, because of an obstacle.

The processing of a pixel block is fairly complicated. We must check for obstacles not only among the pixels one step ahead, where the new pixel block(s) will be located, but also one step outside the new pixel blocks. This will be discussed in more detail in the next chapter.

The pixel blocks must be initiated properly when the object pixels are processed. In a 3·3 neighbourhood, each octant falls between two neighbours. If either or both of the two neighbours are updated from the object pixel, a pixel block for that direction should be created. If there are no obstacles and no other object pixels in the neighbourhood of an object pixel, so that all eight neighbours can be updated, eight pixel blocks, covering a full octant each, are generated.

Obstacles in the neighbourhood will have the most impact on the resulting pixel blocks. A single obstacle pixel will not only prevent itself from being updated, but it will also prevent other neighbours from being updated and causing direction intervals of a number of pixel blocks to be modified or eliminated. This is illustrated in Figure 27.

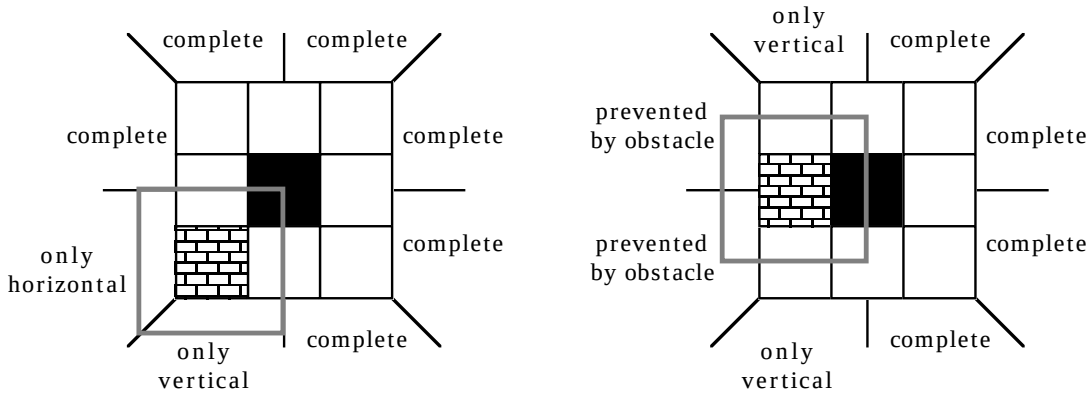


Figure 27. In the first iteration of the algorithm, the object pixels are processed. The figure shows two object pixels (black). One neighbour in each example is an obstacle (grey with a grey border showing the constraints for the moving object center). The pixel blocks close to obstacles get limited direction intervals or are not generated at all.

We even have to test some pixels outside the 3·3 neighbourhood, since they may influence some direction intervals. We will return to this when discussing the processing of a pixel block, where the reasons for this will become apparent.

Circular propagation

When generating non-constrained distance maps with contour processing methods, circular propagation minimizes the execution time. If the propagation contour does not coincide with the equidistance contour (which is circular for Euclidean distances), some areas are unnecessarily updated twice or even more. However, in normal cases this has no significant impact on the execution time.

In [8], circular propagation is achieved by using an upper limit for the distance from the object pixels to the propagation contour. Pixels above the limit are delayed until all other parts of the propagation front have caught up. In [11] the pixels are reached in perfect order of increasing distance using a “bucket” structure. However, the bucket approach is only applicable to non-Euclidean metrics.

If we use pixel blocks, as suggested above, we can no longer use circular propagation. This means that we are facing a loss of speed. However, the problem with multiple entries per pixel in the propagation front, as described above, is much worse than the speed loss from non-coinciding propagation and equidistance contours. Therefore, the algorithm described here will employ pixel blocks but not use circular propagation.

If we want to combine circular propagation with pixel blocks, we could use techniques similar to the ones described in [4] to have pixel blocks along circle segments. When a pixel block is processed, a circle plotting algorithm is used for finding all pixels at a given distance within the direction interval. However, the details of such an algorithm are left for future work.

4. Creating new links, allowing for turns

We now know how to restrict the propagation to line-of-sight. However, we must be able to turn around corners, which means that a new link in the vector chain should be created. Therefore, in addition, each pixel must hold not only the vector that is the next link in a vector chain, but also the total length of the vector chain from that pixel to the object pixel at the end of the chain. If we make a turn, we should create a new link in the vector chain. We must also be certain that new pixel blocks are created along with the new link, hereby supporting propagation in all directions necessary in the shadow of an obstacle.

A new kind of neighbourhood will be used. This neighbourhood consists of three different kinds of neighbours: *forward neighbours, backward neighbours* and *side neighbours*. First, we look at what this means for pixelwise operations.

Definitions: A *forward neighbour* is a pixel within the neighbourhood of the center pixel $\bar{c}$ that satisfies both the following conditions:

- It is located at $\bar{c}-\hat{v}_{major}$ or $\bar{c}-\hat{v}_{major}-\hat{v}_{minor}$
- It falls within the appropriate direction interval.

A *backward neighbour* is a pixel located at $\bar{c}+\hat{v}_{major}$ or $\bar{c}+\hat{v}_{major}+\hat{v}_{minor}$ relative to the center pixel $\bar{c}$. A *side neighbour* is a pixel within the 8-neighbourhood of the center pixel $\bar{c}$ that is neither a forward nor backward neighbour.

Propagation to a forward neighbour takes the vector to the center pixel and adds it to an offset vector. See Figure 29. The vector from the forward neighbour will refer to the same originating pixel as the vector from the center pixel. Propagation to a side neighbour creates a new vector, pointing from the side neighbour to the center pixel, thereby creating a new link in a vector chain.

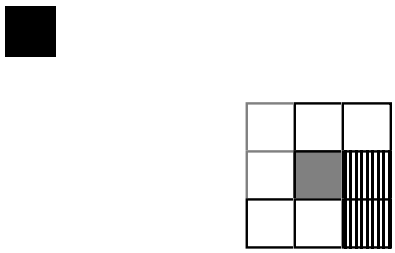


Figure 28. The (grey) center pixel is surrounded by different kinds of neighbours, which are treated differently when the distance map is generated. Striped pixels may be forward neighbours (but not always). White pixels are side neighbours. Backward neighbours (white, grey-border) are not meaningful to include.

The vector chain starting in the forward neighbour will have the same number of links as the vector chain from the center pixel. The vector chain starting in the side neighbour will have one link more than the vector chain from the center pixel.

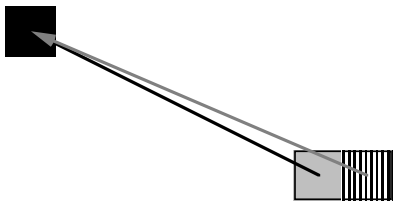


Figure 29. When a forward neighbour (striped) is updated, an offset vector is added to the vector from the center pixel (grey) backwards in the vector chain, producing a longer vector pointing to the same originating pixel.

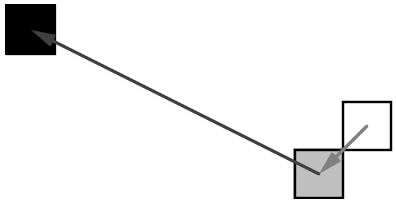


Figure 30. When a side neighbour (white) is updated, it is assigned a vector to the center pixel (grey).

Every time a pixel is processed, we must determine which pixels are forward neighbours and which are side neighbours. We don't need to check the backward neighbours at all, however, for the following reasons.

Theorem: A backward neighbour can never be updated from the center pixel neither as side neighbour nor as a forward neighbour.

Proof: Since the propagation must take place in straight lines, the propagation to the center pixel must have passed through at least one of the two backward neighbours, which therefore must refer to the same originating pixel as the center pixel and can therefore not be updated. If one of the backward neighbours has not been updated to refer to the same originating pixel in an earlier iteration, it is because the propagation to that pixel was cut off by an obstacle, and therefore it should not be updated now, or because it refers to some other originating pixel.

The only case left is if a backward neighbour can be updated as a side neighbour, creating a new link in the vector chain. Denote the backward neighbours $b_1 = \bar{c}+\hat{v}_{major}$ and $b_2 = \bar{c} +\hat{v}_{major}+\hat{v}_{minor}$. When one of the backward neighbours was the center pixel, it had the opportunity to update the other one as a side neighbour.

The path would be o-b_1-b_2 or o-b_2-b_1 as shown in Figure 31. Let the vector from o to c be o-c = (x,y), where x and y can be assumed to be positive (since they refer to the local coordinate system defined by $\hat{v}_{major}$ and $\hat{v}_{minor}$). Then, the length of o-c is ||o-c|| = $(x^2+y^2)^{1/2}$. Thus, we have

$$||o\text{-}c|| = (x^2+y^2)^{1/2}$$

$||c\text{-}b_1|| = ||b_2\text{-}b_1|| = 1$

$||c\text{-}b_2|| = \sqrt{2}$

$||o\text{-}b_2|| = ((x\text{-}1)^2+(y\text{-}1)^2)^{1/2}$

$||o\text{-}b_1|| = ((x\text{-}1)^2+y^2)^{1/2}$

We see that the path o-c-b_1 is longer than o-b_2-b_1 and o-c-b_2 is longer than o-b_1-b_2. Therefore, neither of the backward neighbours can be updated as a side neighbour and does not have to be tested. This completes the proof.

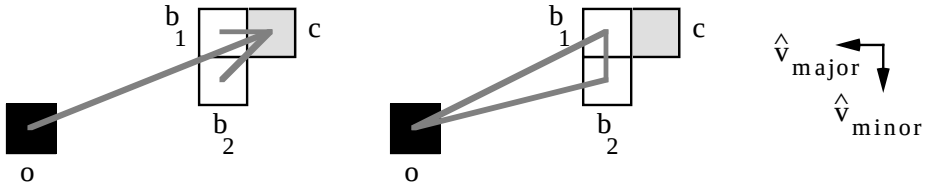


Figure 31. The paths from the originating pixel to b_1 or b_2 through the center pixel c are longer than the paths to either of b_1 or b_2 through the other one.

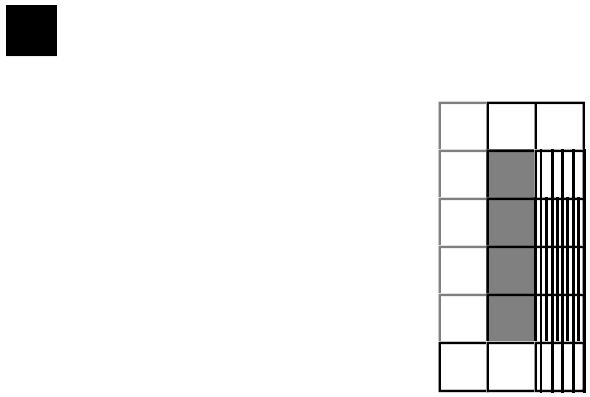


Figure 32. The neighbourhood for a pixel block.

For a pixel block, the neighbourhood is similar, as shown in Figure 32. The shaded pixels in the middle are the center pixels, the actual pixel block. The white pixels are side neighbours. The densely striped pixels are forward neighbours. The sparsely striped pixels can be either forward or side neighbours. Pixels with shaded border are backward neighbours, not interesting to check, and can be excluded from the set of pixels actually included in the processing.

Definitions: A *forward neighbour to a pixel block* is a pixel that is a forward neighbour to any of the pixels within the pixel block. A *backward neighbour to a pixel block* is a pixel that is a backward neighbour to any of the pixels within the pixel block. A *side neighbour to a pixel block* is a pixel that is 8-neighbour to any of the pixels within the pixel block and is neither a forward nor backward neighbour nor a member of the pixel block.

The motivation for excluding the backward neighbours in the pixelwise case (see theorem above) can be extended to the pixel block case.

Let us take a look at an example, Figure 33. A propagation front reaches the point where side neighbours are located within a “shadow”. These side neighbours should be updated from the propagation front (the pixel block in the figure).

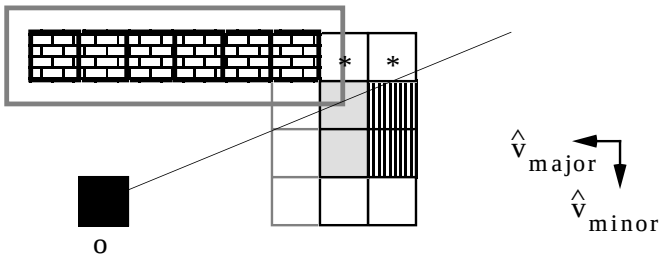


Figure 33. A pixel block (grey) that should update side neighbours (white, solid border), thereby generating new links in the corresponding vector chain. The two side neighbours marked with asterisks (*) are side neighbours in the “shadow” of an obstacle (brick patterned).

For the two side neighbours together, three pixel blocks are generated. See Figure 34. One pixel block is a minimal one, with the direction interval limited to only vertically, straight upwards in the figure. It includes only the pixel closest to the obstacle. This pixel block is not strictly necessary, since the second one will cover the same area and more. The second one includes both side neighbours, and has a direction interval covering the entire octant. The third one includes only the side neighbour farthest from the obstacle and covers a part of an octant. It may cover a full octant, but it is unnecessary.

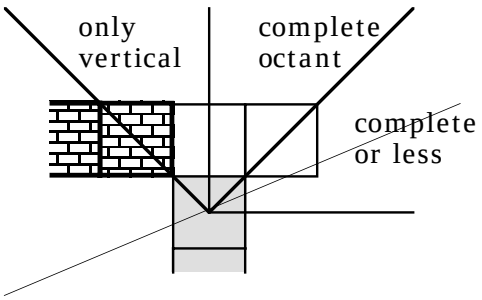


Figure 34. In the example, three pixel blocks are generated at the corner. The thin line shows the edge of the forward region of the original pixel block.

Note that if a direction interval is very narrow, it may be too narrow to hit any pixels at all in forthcoming iterations. In such a case, we might fail to reach some pixels through the optimal path. A slightly longer path will be found for these pixels. These errors have been verified experimentally. We note that they are quite uncommon and do not cause any significant errors.

Basically, this problem is related to the problems noted by Danielsson [1], where it is noted that the scanning EDT is not totally error-free. In both algorithms, the problem is that for some cases the propagation should ideally pass through an area where no grid points exist, since the area is less than one pixel distance wide. Since the propagation of distance values must be done between neighbour grid points, this becomes impossible, and a suboptimal (though very close to optimal) result is found.

With some extra effort, this problem can be solved. However, since the errors are miniscule and do not result in any illegal paths, this is left for future work.

The processing of a pixel block

When processing a pixel block, we must take a number of special cases into account.

The neighbourhood of a pixel block is denoted as shown below. The center pixel at the steepest slope is denoted c_1, while the one at the opposite end is c_2. All pixels that can be side neighbours are given the symbols n_1...n_6. (Note that the pixels n_3 and n_4 can be either forward or side neighbours, depending on the DI.)

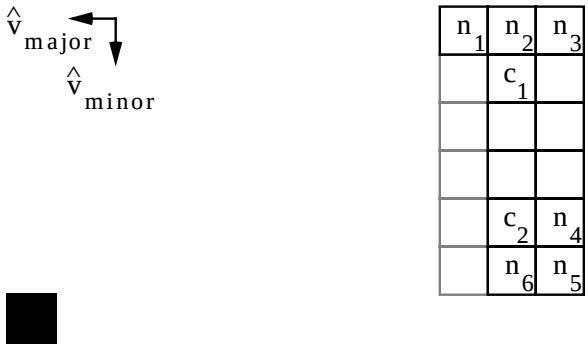


Figure 35. Notation for different members of the neighbourhood of a pixel block.

Before processing the forward neighbours (n_4 to n_3), we must check the pixel closest to the forward neighbours (n_5 in the previous figure) if it is an obstacle pixel. In that case, the direction interval must be modified. This will also occlude the pixel n_4 (unless n_4 is located horizontally or vertically from the originating pixel). See Figure 36.

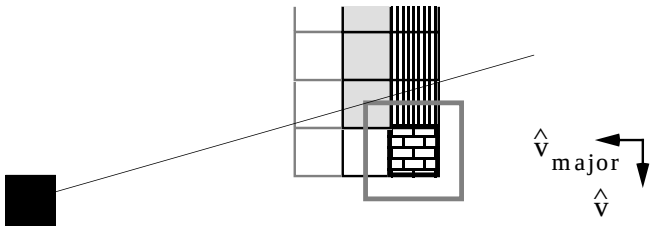


Figure 36. If the pixel below the forward neighbours (striped) of a pixel block (grey) is an obstacle pixel (brick patterned), the direction interval will be reduced to the lower border shown by the line in the figure.

We should process the forward neighbours beginning at the least sloping end, with the pixel n_4 in the figure above, and process every forward pixel in turn until we reach n_3.

See Figure 37. If the other order was used, we would have to check for obstacle pixels one step ahead in every step. If either of n_3 and n_4 are not forward neighbours, they are still checked for being obstacle pixels, but they may not be updated as forward neighbours.

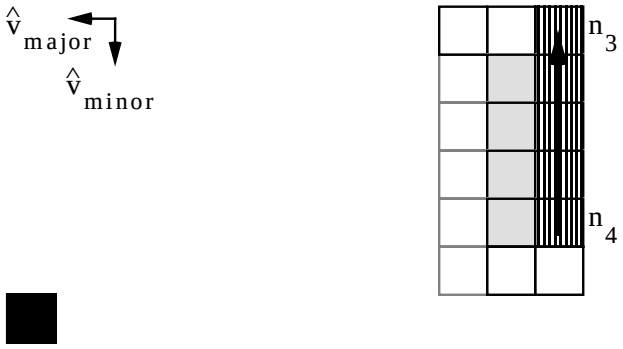


Figure 37. The processing order of the forward neighbours.

When we find an obstacle pixel among the forward neighbours, we must do the following things, illustrated in Figure 38.

I. The direction interval of the pixel block consisting of the previous forward neighbours must be modified.

II. The first non-obstacle pixel must be treated as a side neighbour, since it is hidden behind the obstacle pixel.

III. The direction interval of the next pixel block consisting of other forward neighbours must be modified.

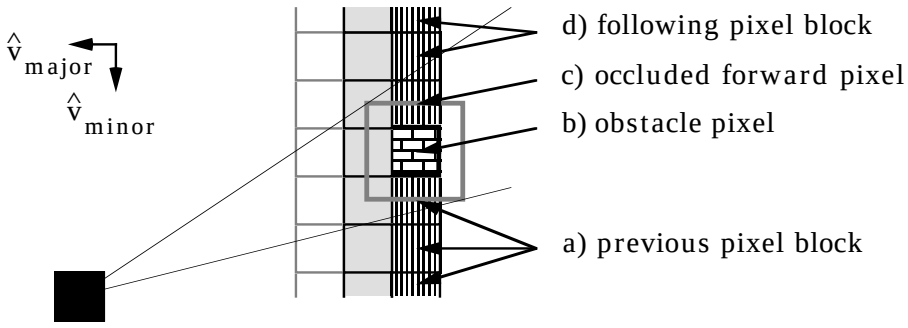


Figure 38. The case where obstacle pixels occur among the forward neighbours. The pixel block (a) consisting of the pixels before an obstacle (b) must have its direction interval reduced to be bordered by the lower line shown. The first pixel after the obstacle (c) must be treated as a side neighbour since it is in the "shadow" of the obstacle. The pixel block consisting of the forward neighbours after that one (d) must have its direction interval reduced to be bordered by the upper line shown.

When we process the side neighbours, we must test some neighbours to make sure that the direction intervals are initiated correctly.

If the side neighbour is a 4-neighbour (n_2, n_4, n_6), we must test two of its neighbours if they are obstacle pixels, namely $\bar{o}$-$\hat{v}_{minor}$ and $\bar{o}$-$\hat{v}_{minor}$-$\hat{v}_{major}$, where $\bar{o}$ is the originating pixel from where the 4-neighbour was updated. This is analogous to the discussion about the processing of the object pixels, above. See Figure 39.

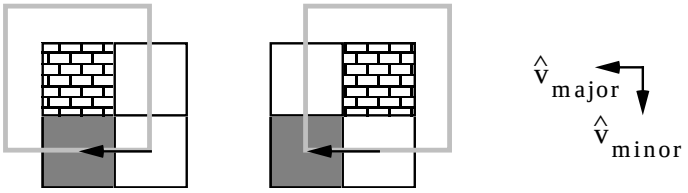


Figure 39. When a 4-neighbour side neighbour is updated (black arrow), we must test if either of the brick patterned pixels are obstacle pixels ($\bar{o}$-$\hat{v}_{minor}$ and $\bar{o}$-$\hat{v}_{minor}$-$\hat{v}_{major}$). In that case, the direction interval will be reduced to only a straight line.

If the pixel is an 8-neighbour but not a 4-neighbour, that is, a diagonal neighbour (n_1, n_3, n_5), we must test the two closest 4-neighbours. If any of them are obstacles, we may not update the pixel at all. This follows from having 8-connected obstacles. We must also test $\bar{o}$-$\hat{v}_{major}$-$2\cdot\hat{v}_{minor}$.

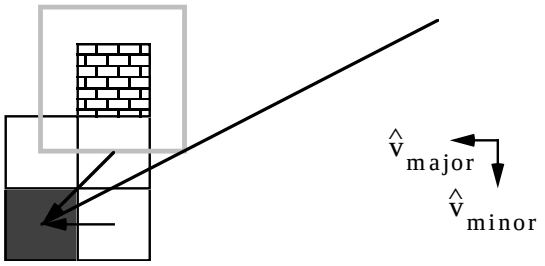


Figure 40. We must also test if the pixel at $\bar{o}$-$\hat{v}_{major}$-$2\cdot\hat{v}_{minor}$ is an obstacle. In that case, the direction interval will be reduced as indicated by the line in the figure.

So far, we have only discussed how direction intervals are influenced by encountering obstacles. When discussing the LDI, it was noted that the LDI regardless of obstacles is a rather small interval for a given pixel and its vector (the one-pixel-out rule mentioned earlier). When only taking obstacles into account, consider the case where two propagation fronts meet, partially eliminating each other. The pixel blocks will be smaller, but the DIs will be unchanged. Thereby, a propagation will not pass some areas, never checking these areas for obstacles, but according to its DI it still is allowed to propagate into areas beyond the unknown areas. Does this mean that we might get vectors that pass straight through obstacles?

We will now make a distinction between the *formal LDI* and the *practical LDI*. The formal LDI is limited by both the one-pixel-out rule and by obstacles, and the practical LDI is only limited by obstacles in the neighbourhood.

See Figure 41. The propagations from two object pixels (black) meet, and they each cover the part of the image above and below the black line (the equidistance contour), respectively. The grey pixels show some pixel blocks generated under the propagation from the lower one of the object pixels within a certain octant. The striped pixels show the DI of each pixel block that does not correspond to the pixel block. A DI may become larger than its pixel block in this way if the practical LDI is used rather than the formal LDI.

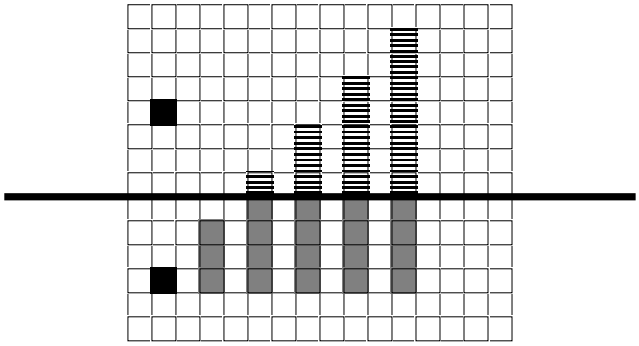


Figure 41. Is the direction interval allowed to reach into areas where the pixel block doesn't reach?

Suppose that the formal LDI is needed for correct results. That would imply that the direction interval must be limited when the propagation front meets another propagation front. That would be necessary if the following problem could occur:

We have two propagation processes P_1 and P_2, originating from the pixels $\bar{o}_1$ and $\bar{o}_2$, respectively. See Figure 43. We also have an obstacle that is located along a straight line from $\bar{o}_1$ to a pixel $\bar{p}$. Then, is it possible that P_1 can

1) reach from $\bar{o}_1$ to $\bar{p}$ without ever touching the obstacle *and*

2) update $\bar{p}$ to point to $\bar{o}_1$?

If that situation is possible and the practical LDI is used, we would get errors, vector pointing across obstacles. We would also be able to eliminate the problems by checking the upper limit of the forward area for each step. However, we can show that this is *not* possible!

Definition: The region *belonging* to the originating pixel $\bar{o}$ is a part of the image where, for all pixels,

- the path to the closest object pixel passes through $\bar{o}$, and
- $\bar{o}$ is the next point where that path turns.

All pixels within the region belonging to $\bar{o}$ *should* have vectors pointing to $\bar{o}$ when the generation of the constrained distance map is finished. See Figure 42.

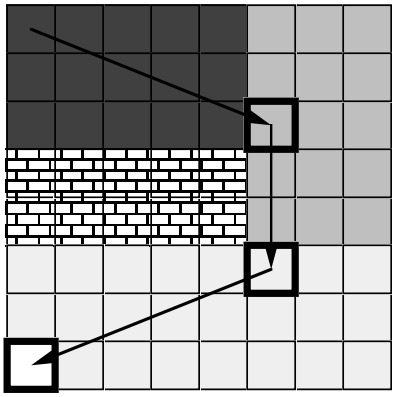


Figure 42. Three regions belonging to three of the pixels (thick border) in a simple constrained Euclidean distance map. The brick-patterned region is an obstacle.

This is not the same as the Voronoi polygons around the object pixels in a non-constrained image. Voronoi polygons are rather the special case of regions belonging to the object pixels in an image with no obstacles.

Typically, the area belonging to $\bar{o}$ is located on one side of $\bar{o}$, the side away from the closest object pixel. Most pixels in a constrained Euclidean distance map will not have any region belonging to it at all. Note that the border between two regions belonging to two different originating pixels are not necessarily a straight line. An area belonging to a pixel is not necessarily convex. The border of the region is the points where the distance to the closest object pixel is the same using the path through $\bar{o}$ as using some other path.

Theorem: If the point $\bar{p}$ is in the region belonging to the originating pixel, all pixels within a straight line from $\bar{p}$ to $\bar{o}$ are also in the region belonging to $\bar{o}$.

Proof: The theorem is easily proved by inspecting the distance values from $\bar{p}$ to two originating pixels. If $\bar{p}$ is closer to one of them, we immediately find that all pixels on a straight line between $\bar{p}$ and the originating pixel are also closer to that originating pixel.

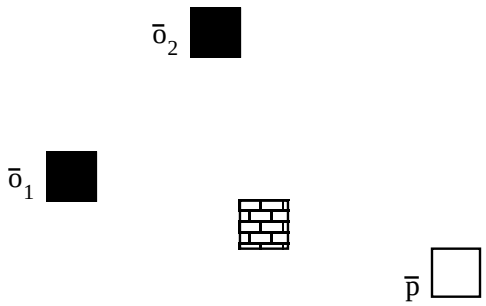


Figure 43. Two originating pixels $\bar{o}_1$ and $\bar{o}_2$, the pixel $\bar{p}$ and an obstacle between $\bar{o}_1$ and $\bar{p}$. We show that it is not possible that $\bar{p}$ is in the region belonging to $\bar{o}_1$ while the obstacle is in the region belonging to $\bar{o}_2$.

Corollary: The DI only have to be decreased when obstacles are encountered, that is, according to the practical LDI.

Proof: The theorem above implies that the propagation P_1 will be able to propagate along a straight line from $\overline{o}$ to $\overline{p}$ if $\overline{p}$ is in the region belonging to $\overline{o}$. Because of the sampling grid, this is true unless the region is too narrow. If so, the propagation will not reach $\overline{p}$ at all, and the error will not occur (though the path from $\overline{p}$ will not be the optimal, only very close to it). If the region is wide enough, P_1 will find any obstacles and limit the direction interval accordingly. (If there is an obstacle between $\overline{o}$ and $\overline{p}$, $\overline{p}$ is naturally not in the region belonging to $\overline{o}$.)

We conclude that the practical LDI may be used instead of the formal LDI, that is, direction intervals only have to be decreased with respect to obstacles. The result is a simplified, faster algorithm, generating the same result as an algorithm using the formal LDI.

Additional data associated with pixel blocks

As noted above, the direction space is partitioned into eight octants. We note that each pixel block need to have its octant number stored explicitly, else we will get problems with pixel blocks consisting of only a single pixel at a diagonal.

A pixel block must also know its originating pixel. If we try to find that pixel by following the vector stored in any of the pixels in the pixel block, we may find the wrong one in the case where another propagation front has reached the pixel block and updated some of the pixels. We can not trust the vectors in the image to be the same when some pixel block is processed, as was stored when that pixel block was created.

Some additional notes on the processing of pixel blocks

A forward neighbour with a vector pointing to the same originating pixel as the processed pixel block may be updated on equal distance value, not only on a lower one, since pixel blocks from one pixel will share pixels along horizontal, vertical and diagonal lines from the originating pixel.

On the other hand, we should not update side neighbours on equal distance. It would be unnecessary, and would sometimes result in multiple pixel blocks propagating over the same area, improving nothing. The previous note makes this particularly important, since two pixel blocks pointing to the same originating pixel may share a pixel, and would create two identical pixel blocks. This would be the case for the two pixel blocks close to the obstacle as shown in figure 34.

Last, we note a simple trick for efficient processing. In each iteration, we should first check all forward neighbours for *all* pixel blocks, and second check all side neighbours for *all* pixel blocks. Last, all the old pixel blocks are removed. This will reduce the number of unnecessary side checks, especially at the edge between two pixel blocks referring to the same originating pixel.

5. The algorithm

In this section, the Constrained Euclidean Distance Transform, the algorithm resulting from the discussions above, is presented and described in detail. First, the data structures to be used are described. Second, we give an outline of the algorithm. Last, the algorithm is described in pseudo-code.

Data structures

We must use a queue or list structure which should hold pointers to the active Contour Set, the propagation front. Each entry of the list holds two pointers, each pointing to one of the two extreme pixels in the pixel block. Each such entry should also hold the maximum and minimum slope, describing the direction interval and thereby what neighbour pixels to treat as forward neighbours.

We also hold information about the general direction, one of eight possible octants, that the pixel block is located in relative to its originating pixel. This information is needed explicitly in the case where a pixel block is consisting of only one pixel, and that pixel is located diagonally, vertically or horizontally from the originating pixel. In other cases, the octant is defined implicitly by other data.

Finally, we should save the location of the originating pixel pointed to by c_1 and c_2 (as defined by Figure 35) at the moment when the pixel block is generated. Later, when the pixel block is processed, the vectors in c_1 or c_2 may be changed by other propagation fronts. Therefore, we should avoid depending on the vectors in c_1 and c_2.

In the image itself, each pixel must hold the vector to the previous link in its vector chain. It should also hold the total distance value. If we didn't keep the distance, we would have to track the whole vector chain each time we need to know the distance value.

We should note that there are some numerical limitations on the precision. There is no apparent way to compare the lengths of two vector chains exactly. The total length of a path is the sum of the length of a number of vectors $\bar{v}_i = (v_{xi}, v_{yi})$.

$$L(Path) = \sum_{i \in Path} \sqrt{v_{xi}^2 + v_{yi}^2}$$

We may use floating point variables for this, but since we know the range (from 1 to the largest distance expected), it should be possible to keep the errors under control and do it much faster with fixed point variables. A N^2 look-up table with integer indexes for square roots is also important for achieving high speed.

The data structures are:

Contour List:
$\bar{c}_1 = (x_1, y_1)$:integer vector

$\bar{c}_2 = (x_2, y_2)$:integer vector
l_{max}: fraction
l_{min}: fraction
z: number of octant
$\bar{n}$: integer vector
Image:
$\bar{v} = (v_x, v_y)$:integer vector (relative)
d: fixed-point

The “fraction” data type is two integers, numerator and denominator of a fraction.

$$l = \frac{n}{}$$

When two fractions are compared, it is done like:

$$l_1 = \frac{n_1}{d_1} > l_2 = \frac{n_2}{d_2} \Rightarrow n_1 \cdot d_2 > n_2 \cdot d_1$$

Since we only use non-zero denominators, this holds with equivalence.

l_{max} and l_{min} are fractions that specify the direction interval within which the “forward neighbours” are located, as described in section 4. The direction of a vector $\bar{v}$ is simply calculated as $\bar{v} \cdot \hat{v}_{minor} / (\bar{v} \cdot \hat{v}_{major})$.

The “fixed-point” data type is a sufficiently long integer, treated as a real number with fixed point. The precision chosen for this data type sets the precision of the whole algorithm. It can be replaced by floating-point variables, but that would slow down the computation of the distance map.

Algorithm outline

This section describe the algorithm briefly, for those of you who do not enjoy pseudo code.

First iteration: The object pixels are processed. Their 8-neighbours are checked and possibly updated and pixel blocks are created in the directions where any neighbour was updated.

Following iterations: Each pixel block from the previous iteration is processed:

Forward neighbours:The pixels n_3 to n_4 including the pixels between are tested for being forward neighbours, that is, if they are within the forward region of the pixel block. All forward neighbours are tested and possibly updated. All consecutive groups of updated pixels form new pixel blocks. The forward regions of the new pixel blocks must fall inside the forward region of the original pixel block.

Side neighbours:The pixels n_1 to n_6, excluding the pixels between n_3 and n_4, are checked for update as side neighbours and pixel blocks are created accordingly.

The algorithm is run until no pixel blocks remain. In the case of path planning where the end points are known, we could introduce another stopping rule. If we determine the minimum distance in the contour in each iteration, we can compare that value to the distance value in all end points. If no end point with a higher value exists, we may stop. However, it introduces even more calculations to the algorithm, and may not be worthwhile.

Pseudo-code algorithm

Below follows a compact description of the algorithm of "pseudo-code" type. The following notation is used:

Pixels in the image are specified by absolute coordinates, and are denoted $\bar{c}_{i = (}x_{ci}, y_{ci})$ for center pixels and $\bar{n} = (x_n, y_n)$ for a neighbour pixel.

The vector in a pixel $\bar{n}$ is found by indexing the image like $\bar{v}[\bar{n}] = \bar{v}[x_n, y_n]$, but is sometimes expressed like $\bar{v}[\bar{n}] = \bar{v}_n = (v_{xn}, v_{yn})$ for briefness. The components of a vector $\bar{v}$ are relative.

The neighbourhood of a pixel block is denoted as in the previous section.

The function $L(\bar{v})$ returns the length of the vector $\bar{v}$, that is $L(\bar{v}) = (v_x^2+v_y^2)^{1/2}$

L_i is the Contour List processed in iteration i.

Pixel block:
$\bar{c}_1 = (x_{c1}, y_{c1})$,
$\bar{c}_2 = (x_{c2}, y_{c2})$,
l_{maxc}, l_{minc},
z_c,
$\bar{o}_c$,
$d_{c1} = d[\bar{c}_1]$, $d_{c2} = d[\bar{c}_2]$
Neighbour pixel:
$\bar{n} = (x_n, y_n)$,
$\bar{v}_n = (v_{xn}, v_{yn}) = \bar{v}[\bar{n}]$,
$d_n = d[\bar{n}]$
New pixel block:
$\bar{b}_1$, $\bar{b}_2$
l_{maxb}, l_{minb},
z_b,
$\bar{o}_b$
Suggested values for the neighbour pixel:
$\bar{v}_s = (v_{xs}, v_{ys})$,
d_s

Other:
$\bar{e}$: offset vector from center pixel to neighbour.
open: a logical variable
l_{mint}: a temporary variable for the lower direction bound

From z_c, we can get the vectors $\hat{v}_{major}$ and $\hat{v}_{minor}$, horizontal or vertical unit vectors, different for each octant as shown in figure 44. With these, the pixels $\bar{s}_1$...$\bar{s}_6$ can be expressed as functions of $\bar{c}_1$ and $\bar{c}_2$. For example, $\bar{s}_1 = \bar{c}_1 - \hat{v}_{major} + \hat{v}_{minor}$. Whenever any of the pixels $\bar{s}_1$...$\bar{s}_6$ is mentioned, a sum of $\bar{c}_1$, $\bar{c}_2$, $\hat{v}_{major}$ and $\hat{v}_{minor}$ is implied. The numbering of octants used for z_c is shown in the figure below.

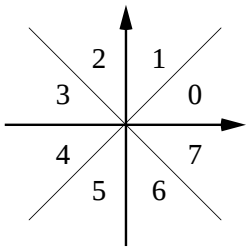


Figure 44. The numbering of octants used for the variables z.

Initialization:

All *obstacle pixels* get the distance value d=0 and a vector $\bar{v}$ = (0,0)

All *background pixels* get the distance value d=∞ and a vector $\bar{v}$ = (0,0)

All *object pixels* (typically only one or very few) get the distance value d=0 and a vector $\bar{v}$ = (0,0).

Algorithm:

[First, process the object pixels]

```
for all object pixels c̄:
    side-check1(c̄-(1,0),(1,0),1,-1, 0)                    [side-check1 checks the 4-]
    side-check2(c̄-(1,1),(1,1),√2,0, 1)                    [neighbours, side-check2 the]
    side-check1(c̄-(0,1),(0,1),1,1, 2)                     [8-but-not-4-neighbours]
    side-check2(c̄-(-1,1),(-1,1),√2,2, 3)
    side-check1(c̄-(-1,0),(-1,0),1,3, 4)
    side-check2(c̄-(-1,-1),(-1,-1),√2,4, 5)
    side-check1(c̄-(0,-1),(0,-1),1,5, 6)
    side-check2(c̄+(1,-1),(-1, 1),√2,6, 7)
    if (1,0) = v̄[c̄ - (1,0)]                               [if the first side-check1 succeeded]
        if open
            b̄2 := c̄ - (1,0)
        else
            Create new entry in L1
            open := true
            b̄1 := c̄ - (1,0)
```

$\bar{b}_2 := \bar{c}$ - (1,0)
$z_b := 7$
$\bar{o}_b = \bar{c}$
if d[$\bar{c}$ - (1,-1)] = 0 or d[$\bar{c}$ - (0,-1)] = 0 *[obstacle in $\bar{c}$ - (1,-1) or $\bar{c}$ - (0,-1)]*
    $l_{maxb} := 0/1$
else
    $l_{maxb} := 1/1$
$l_{minb} := 0/1$

[The main loop - check an entry at a time.]
while L_i is not empty
[First, check all forward neighbours and generate new pixel blocks]
for all entries in L_i
get $\bar{c}_1$, $\bar{c}_2$, l_{maxc}, l_{minc}, z_c, $\bar{o}$ from L_i.
open := false
$l_{mint} := l_{minc}$
set $\hat{v}_{major}$ and $\hat{v}_{minor}$ according to z_c
if d[$\bar{o}$-$\bar{c}_2$+$\hat{v}_{major}$-$\hat{v}_{minor}$]=0 *[test for obstacle]*
    $l_{mint} := \max\{l_{mint}, (\bar{v}_s \cdot \hat{v}_{minor} + 1)/(\bar{v}_s \cdot \hat{v}_{major} - 1)\}$
for $\bar{n} := \bar{s}_4$ to $\bar{s}_3$, step -$\hat{v}_{minor}$ *[a range of pixels along $\hat{v}_{minor}$]*
    $\bar{v}_s := \bar{o}_c - \bar{n}$
    if d[$\bar{n}$] = 0 and open=true *[next forward neighbour is an obstacle]*
        $l_{maxb} := \min\{l_{maxc}, (\bar{v}_s \cdot \hat{v}_{minor} - 1)/(\bar{v}_s \cdot \hat{v}_{major} + 1)\}$ *[obstacle corner]*
    $d_s = L(\bar{v}_s) + d[\bar{o}_c]$
    if ($d_s < d_n$ or ($d_s \le d$ and $\bar{v}_n = \bar{v}_s$))
            and $l_{mint} < \bar{v}_s \cdot \hat{v}_{minor}/(\bar{v}_s \cdot \hat{v}_{major}) < l_{maxc}$ *[note l_{mint}!]*
        $\hat{v}_n := \hat{v}_s$
        $d_n := d_s$
        if not open
            Create new entry in L_{i+1}
            $\bar{b}_2 := \bar{n}$
            $z_b := z_c$
            $\bar{o}_b = \bar{o}_c$
            $l_{minb} := l_{mint}$ *[DI of the new pixel block]*
            open := true
        $\bar{b}_1 := \bar{n}$
        $l_{maxb} := l_{maxc}$ *[DI of the new pixel block]*
    else *[$\bar{n}$ is not updated]*
        if d_n=0 *[no update because of obstacle, adjust distance interval]*
            if open
                $l_{maxb} = \min\{l_{maxc}, (\bar{v}_s \cdot \hat{v}_{minor} - 1)/(\bar{v}_s \cdot \hat{v}_{major} + 1)\}$ *[obstacle corner]*
            $l_{mint} = \max\{l_{mint}, (\bar{v}_s \cdot \hat{v}_{minor} + 1)/(\bar{v}_s \cdot \hat{v}_{major} - 1)\}$ *[obstacle corner]*
        else *[a pixel that is in a "shadow"]*
            open := false
            side-check1[$\bar{n}$, $\hat{v}_{major}$, 1 + d[$\bar{n}$ + $\hat{v}_{major}$], getz(0), getz(7)
        open := false
[Second, check all side neighbours and create new pixel blocks in other directions]

for all entries in L_i
get $\bar{c}_1$, $\bar{c}_2$, z_c from L_i.
open := false
side-check2($\bar{s}_1$, $\bar{c}_1$ - $\bar{s}_1$,$\sqrt{2}$ + d_{c1},getz(3), getz(2))
side-check1($\bar{s}_2$, $\bar{c}_1$ - $\bar{s}_2$,1 + d_{c1},getz(2), getz(1))
side-check2($\bar{s}_3$, $\bar{c}_1$ - $\bar{s}_3$,$\sqrt{2}$ + d_{c1},getz(1), getz(0))
if $\bar{c}_1 \neq \bar{c}_2$ open := false
side-check1($\bar{s}_4$, $\bar{c}_2$ - $\bar{s}_4$,1 + d_{c2},getz(0), getz(7))
side-check2($\bar{s}_5$, $\bar{c}_2$ - $\bar{s}_5$,$\sqrt{2}$ + d_{c2},getz(7), getz(6))
side-check1($\bar{s}_6$, $\bar{c}_2$ - $\bar{s}_6$,1 + d_{c2},getz(6), getz(5))
delete L_i
[end of algorithm]

[subroutines]

getz(t) = { z_c odd: (z_c + 8 - t) mod 8
z_c even: (z_c + t) mod 8

[side-check 1, used for testing diagonal (8-but-not-4-)neighbours]

side-check1($\bar{n}$, $\bar{v}_s$, d_s, z_{old}, z_{new})
if $d_s < d_n$
$\hat{v}_n := \hat{v}_s$
$d_n := d_s$
if $z_{old} < 0$ *[z < 0 : don't create any pixel block!]*
set $\hat{v}_{minor}$ according to z_{old} *[$\hat{v}_{major} = \bar{v}$]*
if not open
Create new entry in L_{i+1}
open := true
$\bar{b}_1 := \bar{n}$
$z_b := z_{old}$
if d[$\bar{n}$-$\hat{v}_{minor}$]=0 or d[$\bar{n}$-$\hat{v}_{minor}$ + $\bar{v}$]=0 *[check for obstacles]*
l_{maxb} := 0/1
else
l_{maxb} := 1/1
l_{minb} := 0/1
else [if open]
if d[$\bar{n}$-$\hat{v}_{minor}$]=0 or d[$\bar{n}$-$\hat{v}_{minor}$ + $\bar{v}$]=0 *[check for obstacles]*
l_{maxb} := 0/1
else
l_{maxb} := min{l_{maxb}, 1/1}
l_{minb} := max{l_{minb}, 0/1}
$\bar{b}_2 := \bar{n}$
$\bar{o}_b := \bar{n} + \bar{v}_s$
Create new entry in L_{i+1}
set $\hat{v}_{minor}$ according to z_{new} *[$\hat{v}_{major} = \bar{v}$]*
open := true
$\bar{b}_1 := \bar{n}$

$\bar{b}_2 := \bar{n}$
$z_b := z_{new}$
$\bar{o}_b := \bar{n} + \bar{v}_s$
if $d[\bar{n}\text{-}\hat{v}_{minor}]=0$ or $d[\bar{n}\text{-}\hat{v}_{minor} + \bar{v}]=0$ *[check for obstacles]*
    $l_{maxb} := 0/1$
else
    $l_{maxb} := 1/1$
$l_{minb} := 0/1$
else *[$\bar{n}$ is not updated]*
open := false

[side-check 2, used for testing 4-neighbours]

side-check2($\bar{n}$, $\bar{v}_s$, d_s, z_{old}, z_{new})
if $d_s < d_n$ and $d[\bar{n}+(v_x,0)]\neq 0$ and $d[\bar{n}+(0,v_y)]\neq 0$ *[note the obstacle checks!]*
    $\hat{v}_n := \hat{v}_s$
    $d_n := d_s$
    set $\hat{v}_{minor}$ according to z_{old} *[$\hat{v}_{major}$ not needed]*
    if not open
        Create new entry in L_{i+1}
        open := true
        $\bar{b}_2 := \bar{n}$
        $z_b := z_{old}$
        $l_{minb} = 0/1$
        if $d[\bar{n}\text{-}\hat{v}_{minor}]=0$
            $l_{maxb} = 1/2$
        else
            $l_{maxb} = 1/1$
    else *[if open]*
        $l_{minb} = \max\{l_{minb}, 0/1\}$
        if $d[\bar{n}\text{-}\hat{v}_{minor}]=0$
            $l_{maxb} = \min\{l_{maxb}, 1/2\}$
        else
            $l_{maxb} = \min\{l_{maxb}, 1/1\}$
    $\bar{b}_1 := \bar{n}$
    $\bar{o}_b := \bar{n} + \bar{v}_s$
    Create new entry in L_{i+1}
    set $\hat{v}_{minor}$ according to z_{new} *[$\hat{v}_{major}$ not needed]*
    open := true
    $\bar{b}_1 := \bar{n}$
    $\bar{b}_2 := \bar{n}$
    $z_b := z_{new}$
    $\bar{o}_b := \bar{n} + \bar{v}_s$
    $l_{minb} = 0/1$
    if $d[\bar{n}\text{-}\hat{v}_{minor}]=0$
        $l_{maxb} = 1/2$
    else

$l_{maxb} = 1/1$

else *[$\bar{n}$ is not updated]*

open := false

Examples

In this section, we show a number of images with obstacles and the constrained maps resulting from applying the algorithm on these images.

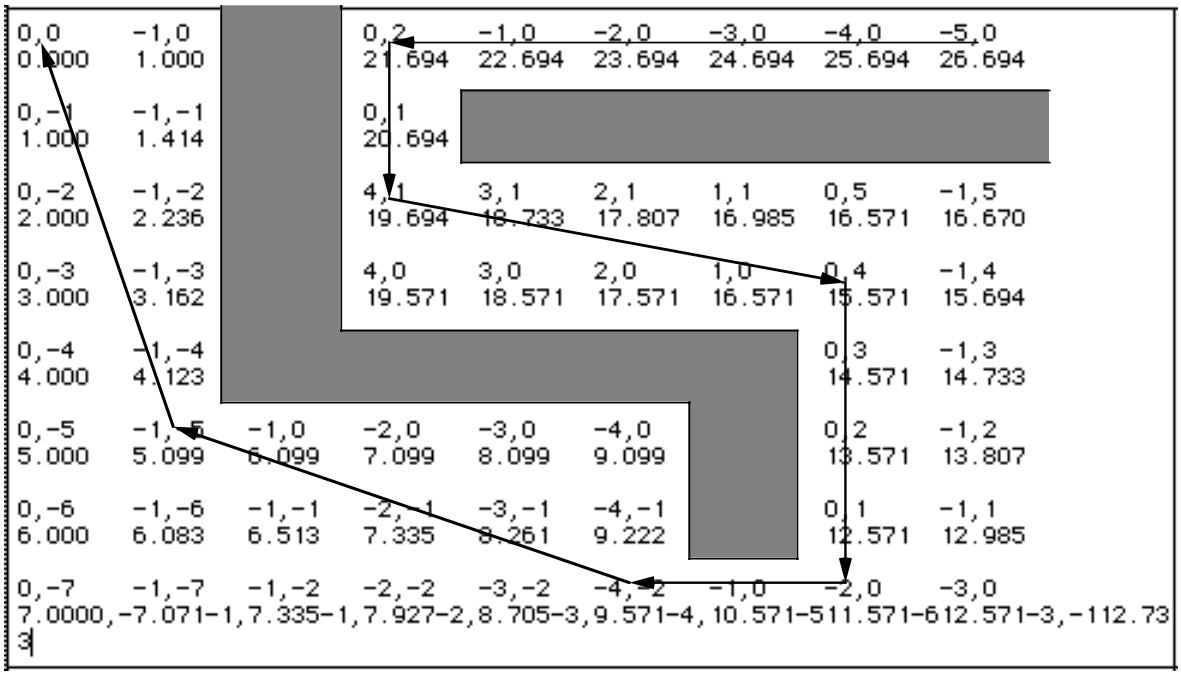


Figure 45. The constrained Euclidean distance map of a very small image. Obstacles are grey. For each pixel, the vector and the distance value is shown. Note that the origin of the coordinate system is in the upper left corner, and that the x axis is pointing downwards.

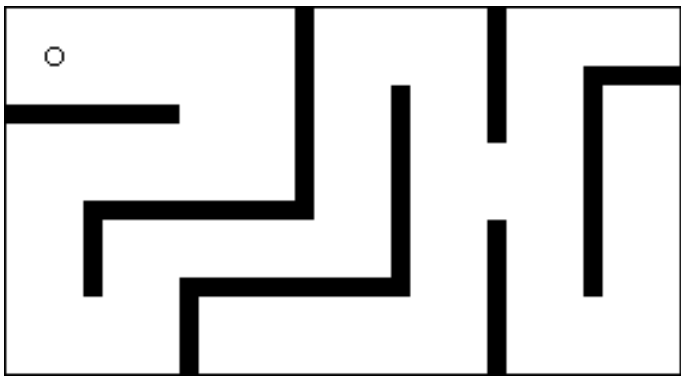


Figure 46. An image with obstacles (black) and a single object pixel (circle).

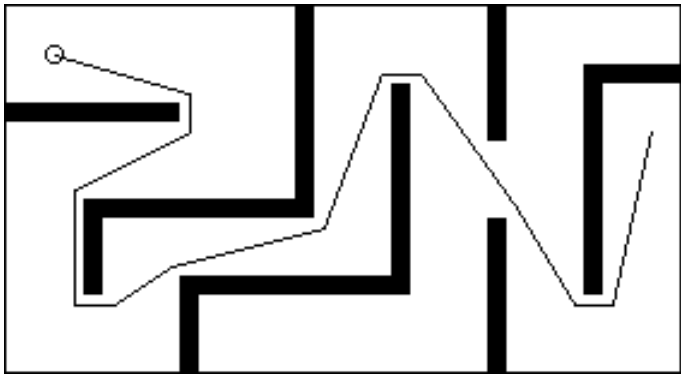


Figure 47. One of the vector chains generated by applying the Constrained Euclidean distance transform on the image in the previous figure. (The vectors are slightly jagged, since the image is cut from a screen dump, and therefore is bit-mapped. In reality they are, of course, absolutely straight.)

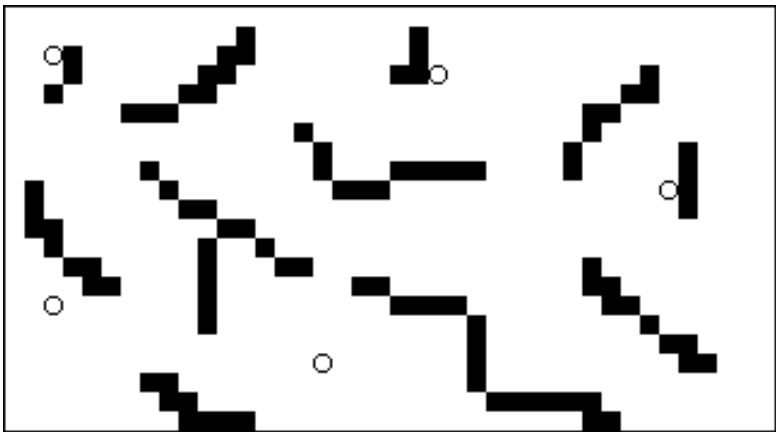


Figure 48. An image with obstacles (black) and a a number of object pixel (circles).

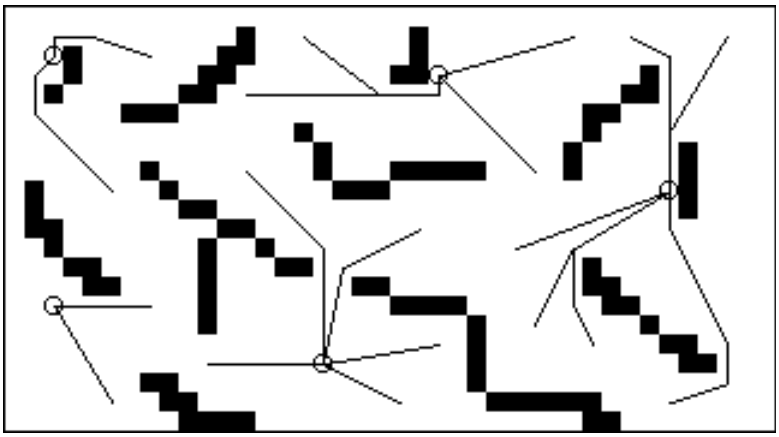


Figure 49. Some of the vectors and vector chains generated by applying the Constrained Euclidean distance transform on the image in the previous figure.

6. A simplified algorithm

In this section, a simplified version of the Constrained Euclidean Distance Transform is presented. This version gives larger and more frequent errors than the one presented above, but it still generates legal paths, optimal within a pixel distance. This version is included mainly to illustrate the variety of options available when implementing a constrained Euclidean distance transform and is not a recommended algorithm for practical use. Hence, the entire algorithm will not be described in detail here.

If we use the following definitions for obstacles and the moving object instead for the ones made earlier in this paper, the algorithm will be a lot simpler.

Definition: A single obstacle pixel is point-shaped, side length zero.

Definition: The object that is to be moved through the image is point-shaped, side length zero.

Definition: The obstacles are 4-connected. A path may not pass between two connected obstacle pixels.

Definition: The background is 8-connected. This follows from the previous definition.

Normally, we would want obstacle pixels to be larger than a point, typically covering a square, like in the earlier definition. We can, however, easily transform a “square pixel” image to a “point pixel” image simply by expanding the obstacles one step “south” followed by one step “east”, as shown in the figure below. This will map each obstacle pixel to four pixels in a square. Apart from having the grid translated one-half pixel distance horizontally and vertically, the two representations are equivalent for a single object pixel. Beware, however, that this transformation may alter the topology of the image, since objects can be joined in the operation. 8-connected obstacles become 4-connected and 1 pixel wide cracks disappear.

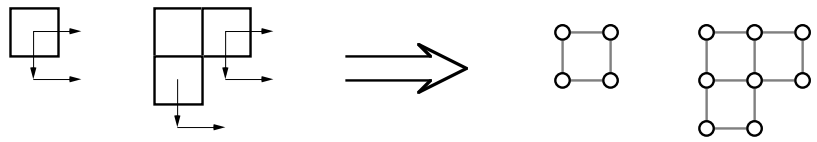


Figure 50. A simple transformation to turn an image with square-shaped pixels into an image with point-shaped ones.

With these definitions, we don't have to test for obstacles as before. We just have to keep track of the maximum distance interval, the LDI. When an obstacle is encountered, the obstacle pixels will not be updated. The pixel blocks generated that cause propagation beyond the obstacle will get direction intervals limited by the LDI of the pixels used.

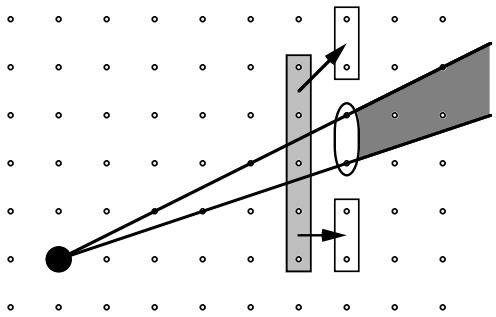


Figure 51. When an obstacle is encountered, the pixel blocks generated on each side of the obstacle will have direction intervals that is limited by the edges of the obstacle. With point-sized obstacle pixels, these limits are easily found as the obstacle pixels themselves.

With these definitions of the moving object and the obstacles, we can avoid obstacles by using the *formal LDI*, as mentioned previously. We do not have to check for obstacles in neighbours of the pixels that we check. We still may not update an obstacle pixel, of course. If the obstacle pixels hold the distance value zero, we do not have to make any distinction between obstacles and other pixels that should not be updated.

The formal LDI may be introduced by the following change:

$l_{minb} := l_{mint}$ *[DI of the new pixel block]*
open := true
$\bar{b}_1 := \bar{n}$
$l_{maxb} := l_{maxc}$ *[DI of the new pixel block]*

is changed to

$l_{minb} := \max(l_{minc}, (\bar{v}_s \cdot \hat{v}_{minor} - 1)/(\bar{v}_s \cdot \hat{v}_{major}))$ *[DI of the new pixel block]*
open := true
$\bar{b}_1 := \bar{n}$
$l_{maxb} := \min(l_{maxc}, (\bar{v}_s \cdot \hat{v}_{minor} + 1)/(\bar{v}_s \cdot \hat{v}_{major}))$ *[DI of the new pixel block]*

A large number of obstacle checks should be removed.

If we modify the algorithm described above according to these new definitions, the algorithm will be simpler, with far less special cases. However, the algorithm will quite often return suboptimal paths. The reason is that the corners, where the paths should make turns, are no longer unique. See Figure 52. The path will turn in different pixels depending on the direction of the path. The result is a close approximation to the optimal path.

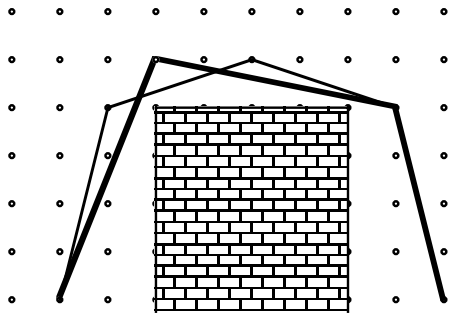


Figure 52. With the definitions above, the corners are no longer uniquely defined. The bold path is the one found by the simplified algorithm. The thin path is the shortest path.

This will cause errors as large as $\sqrt{2}$-1 pixel distance in a single turn, but still the algorithm holds most of the features of the algorithm suggested previously. The errors are reduced if resolution is increased, which is not the case with non-Euclidean algorithms.

It is questionable whether this algorithm has any significant speed advantage over the algorithm described above. In an experimental implementation, hardly any speed difference was observed. The algorithm could be of some interest for systems where the cost of accessing the image is significantly larger than arithmetic operations.

7. Conclusions

In this paper, we have presented the Constrained Euclidean Distance Transform, an algorithm for generating Euclidean distance maps in 2-dimensional images sampled with the Cartesian grid, in which obstacles are allowed. The algorithm can be used for finding the shortest path between two points in an image with obstacles.

Other algorithms for this problem generally belong to one of two classes. The algorithms in the first class operate on sampled images and find paths that strictly follows the sampling grid, moving from some pixel in a path only to one of its neighbours. Algorithms doing this in constant time ($O(N^2)$) for constant image size exist. The algorithms in the other class operate on a parametric description of the obstacles, typically polygonal, and generate path descriptions in the form of a chain of vectors. The execution time of such algorithms are very data-dependent. The Constrained Euclidean Distance Transform differs from older algorithms in that it operates on a sampled image, but the resulting path is a number of vectors.

The Constrained Euclidean Distance Transform has the following features:

- High precision, exacts results in almost all cases.
- The processing time is proportional to the number of pixels in the image.
- 1 pixel wide obstacles are allowed.
- Compact, easy-to-use output; a vector chain.

The algorithm is a contour processing algorithm, where the propagation front is stored in a list, the Contour List. Along with the pointers to the pixels in the propagation front, other data is stored in the Contour List. The most important information added is the *allowed direction interval*, which describes the area where the actual propagation is allowed to take place in a straight line. Each index in the Contour List describes not only one pixel in the propagation front, but rather a number of consecutive pixels in the propagation front, a *pixel block*.

Possible extensions to the algorithm include:

- higher dimensions, finding paths through 3D-space or allowing the object to be moved to rotate.
- making the algorithm totally error-free.
- speeding up the algorithm by using circular propagation fronts.

References

[1] P.E. Danielsson, “Euclidean Distance Mapping”, *Computer Graphics and Image Processing* 14, 1980, pp 227-248.

[2] L.J. van Vliet and B.J.H Verwer, “A contour processing method for fast binary neighbourhood operations”, *Pattern Recognition Letters* 7, 1988, pp 27-36.

[3] Q.Z. Ye, “The Signed Euclidean Distance Transform and Its Applications”, *Proceedings, 9:th International Conference on Pattern Recognition*, 1988, pp 495-499.

[4] Z. Kulpa, B. Kruse, “Algorithms for circular propagation in discrete images”, *Computer Vision, Graphics and Image Processing* 24, 1983, pp 305-328.

[5] D. T. Lee, F.P. Preparata, “Euclidean Shortest Paths in the Presence of Rectilinear Barriers”, *Networks* 14, 1984, pp 393-410.

[6] L. Dorst and P.W. Verbeek, “The constrained distance transformation: A pseudo-Euclidean, recursive implementation of the Lee-algorithm”, *Signal Processing III: Theories and Applications* (Proc. European Signal Processing Conf.), I.T. Young et. al., eds, Elsevier Science Publ. B.V., 1986, pp 917-920.

[7] I. Ragnemalm, “The Euclidean Distance Transform and its implementation on SIMD architectures”, *Proceedings, 6th Scandinavian Conf. on Image Analysis*, 1989, pp 379-384.

[8] I. Ragnemalm, “Contour processing distance transforms”, in: Cantoni et. al., eds., *Progress in Image Analysis and Processing*, World Scientific, Singapore, 1990, pp 204-212.

[9] J. Piper, E. Granum, “Computing Distance Transformations in Convex and Non-convex Domains”, *Pattern Recognition* 20, 1987, pp 599-615.

[10] U. Montanari, “A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance”, *Journal of the ACM* 15, 1968, pp 600-624.

[11] B.J.H. Verwer, “Improved metrics in Image Processing applied to the Hilditch Skeleton”, *Proceedings, 9:th International Conf. on Pattern Recognition*, 1988, pp 137-142.

[12] B.J.H. Verwer, P.W. Verbeek and S.T. Dekker, “An Efficient Uniform Cost Algorithm Applied to Distance Transforms”, *IEEE Trans. on Pattern Analysis and Machine Intelligence* 11, 1989, pp 425-429.

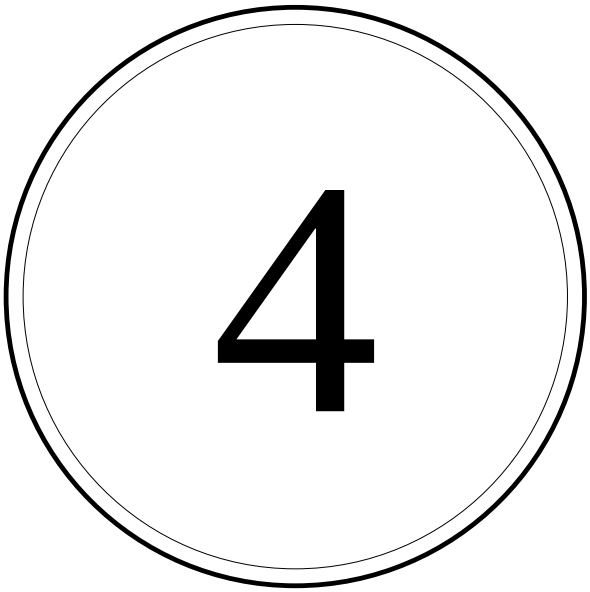
[13] A. Rosenfeld, J. Pfaltz, “Sequential Operations in Digital Picture Processing”, *Journal of the ACM* 13, 1966), pp 471-494.

[14] G. Borgefors, “Distance Transformations in Digital Images”, *Computer Vision, Graphics and Image Processing* 34, 1986, pp 344-371.

[15] J.T. Schwartz et. al. (ed), *Planning, Geometry, and Complexity of Robot Motion*, Ablex Publishing Corporation, ISBN 0-89391-361-8.

[16] P.E. Danielsson, S. Tanimoto, “Time complexity for serial and parallel propagation in images”, *Proc. SPIE vol. 435, Architecture and Algorithms for Digital Image Processing*, 1983, pp 60-67.

[17] G. Borgefors, “Distance Transformations in Arbitrary Dimensions”, *Computer Vision, Graphics and Image Processing* 27, 1984, pp 321-345.



Fast erosion and dilation by contour processing and thresholding of distance maps

Pattern Recognition Letters 13, 1992.
Old version in: Proceedings, 7th Scandinavian Conference on Image Analysis, Aalborg, 1991.

Fast erosion and dilation by contour processing and thresholding of distance maps

Ingemar Ragnemalm

Dept. of Electrical Engineering, Linköping University, S-581 83 Linköping, Sweden

Abstract:

This paper describes efficient algorithms for performing propagation (i.e. erosion and dilation). The proposed algorithm performs ordered propagation using Euclidean distance transformation without generating any distance map. This allows optimization of both the time and memory demand.

Key words: Distance transformation, propagation, erosion, dilation, contour processing, ordered propagation, binary image processing, Euclidean metric.

1. Introduction

Erosion and dilation are the two fundamental operations of mathematical morphology. Their applications include suppression of salt-and-pepper-noise, smoothing of jagged edges, PCB inspection and clustering (e.g. Rosenfeld (1976)). In this paper, we present new algorithms for performing these operations with higher speed.

Basic operations:

The dilation of X (the image) by B (the structural element) is written $X \oplus B$. The erosion of X by B is written $X \ominus B$. Erosion and dilation are dual operations. Denoting the complement of X as X^c, erosion and dilation has the following relations:

$$X \ominus B = (X^c \oplus B)^c X \oplus B = (X^c \ominus B)^c$$

Because of this duality, algorithms in this paper are written as dilation algorithms. The erosion version is easily derived by dilating X^c. For formal definitions and other theorems, see Dougherty (1987) and Serra (1982).

Variables used in this paper:

Propagation distance **d** is the radius of the structure element B. In this text, B is always symmetrical and circular or approximately circular. Thus, the erosion/dilation operation moves the border of all objects in the image the distance **d**.

Image size is **n·n**. This is the number of pixels in the binary source image.

Influenced area **A** is the total area (number of pixels) influenced by the erosion or dilation operation.

Active contour length **l** is the border length at some moment during the operation.

Maximum contour length $\mathbf{l_{max}}$ is the maximum value of **l** at any moment during the operation.

In a sampled image, an n·n array, the influenced area **A** is the same as the number of influenced pixels. Here, **l** is the number of pixels along the borders at some stage in the operation and **d** is the number of pixel distances to propagate. The symbols are illustrated in figure 1.

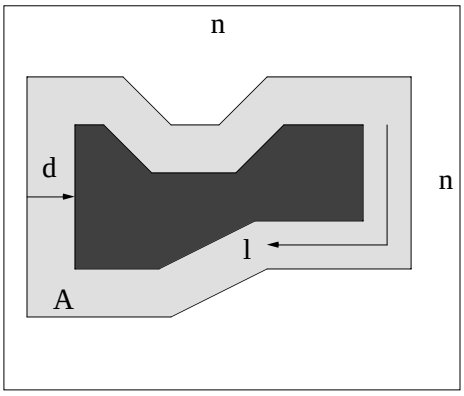


*Figure 1. The erosion operation erodes a binary image from the total shaded area to the dark shaded area. The image size is **n·n** pixels. The distance between the outer (initial) border and the inner (resulting) border is **d** pixel distances.*

Traditionally, erosion and dilation is performed with morphological operators applied in parallel over the whole image, as described by Rosenfeld (1982). The computational complexity is in $O(n^2 \cdot d)$. A more efficient method for sequential computers is to perform a distance transformation on the binary image (e.g. Laÿ (1987)). Thresholding the resulting distance map at the appropriate distance value results in the desired eroded or dilated binary image. The operation has a computational complexity in $O(n^2)$. Van Vliet (1988) suggested an even more efficient method, namely contour processing. The propagation front is stored in a list, and only the propagation front is processed in each iteration. The computational complexity for contour processing erosion or dilation is in O(A), but the initialization demands a scan over the entire image, which is an operation in $O(n^2)$. However, only simple metrics like City Block or Chessboard are supported by this method.

Table 1 gives an overview over the algorithms mentioned above. We assume that a single-processor system is used. With parallel hardware, both parallel and distance map based algorithms can be speeded up.

The memory demand does not include the memory space occupied by the binary source and destination images. The computation time for Contour Processing does not include the initialization step, which of course is an $O(n^2)$ operation with a very low constant factor.

Algorithm	Computation	Memory	Image scans
Parallel	$O(n^2 \cdot d)$	0	d
Scanning DT	$O(n^2)$	$O(n^2)$	fixed
Contour proc.	O(A)	$O(l_{max})$	1

Table 1. Comparison of some common erosion or dilation algorithms by computational complexity and memory demand.

2. Distance transformations

A *distance transformation* generates a greyscale image from a binary image, where each pixel in the resulting greyscale image, the *distance map*, holds the distance to the closest set (*object* or *feature*) pixel in the binary image. Usually, algorithms using integer arithmetics are used (e.g. Borgefors (1986)).

A special kind of distance map is the *Euclidean distance map*, generated by a *Euclidean distance transformation*, as suggested by Danielsson (1980). In this case, the resulting image holds vectors rather than scalar distance values. Each vector points to the closest object pixel.

Distance maps are usually generated by neighbourhood operations. We distinguish three different classes of algorithms: parallel, raster-scanning and contour processing algorithms.

In this paper we will use contour processing algorithms. Contour processed distance transforms in its basic form has been suggested by Piper and Granum (1987), which performs propagation with square propagation front. Verwer (1989) used bucket sorting to process the propagation front in perfect order. A simple method for achieving approximately ordered propagation as well as algorithms for Euclidean distance mapping was suggested by Ragnemalm (1990).

3. Contour processed distance transforms over limited distance

Since distance map generation can be speeded up by contour processing, erosion and dilation by thresholding of distance maps can also be speeded up by using contour processing distance transformations (not to be confused with the contour processed erosion and dilation, mentioned in section 1). In this case, however, the performance can be improved considerably by halting the propagation of distance values after a number of iterations. With circular propagation, only the areas that will be influenced by the erosion or dilation will be processed.

The distance transformation has a computational complexity in O(A). The distance map must be initiated and thresholded, each having computational complexity in $O(n^2)$. These terms have very small constant factors, and are insignificant in algorithms generating complete distance maps. In this case, where we intend to cover only a small part of the image, we should pay some attention to the efficiency of these steps.

In the initialization, we only put object pixels on the contour list that have at least one background pixel as a 4-neighbour, rather than queuing all object pixels.

The efficiency may be improved further by embedding the thresholding in the propagation process. When a pixel is given a distance value below the threshold, we may set the corresponding binary pixel immediately. For scanning distance transformations, this is of no significance since all pixels are given distance values and therefore tested. For a contour processed distance transform over limited distance, however, the thresholding operation is reduced to an operation in O(A), just like the distance mapping operation that it is now a part of.

4. Contour processed distance transformation with no distance map

So far, we have stated that erosion and dilation can be performed in short time using distance transformations. However, this introduces a higher demand for memory in the form of a complete greyscale (or vector) image. In this section we introduce a new method that combines high speed with lower memory demand.

Distance information is included in the propagation front list (contour list), and the algorithm operates directly on the binary image. A distance transformation is performed, but no distance map is generated.

This method demands perfectly ordered propagation in the sense that the pixels must be accessed in increasing distance order. The binary image will be updated during the propagation process. In each stage of the operation, the binary value of each pixel in the binary image will tell if it is in or outside the region processed so far. Thus, it tells if it is on a higher or lower distance from the object pixels than the propagation front. When the propagation front reaches the erosion/dilation distance **d**, the operation is finished. Note that the contour list holds pixels that have not been processed yet.

An important problem is how to access the pixels in perfect order. For integer distance values, including Chamfer distance, this can be solved by using bucket sorting. The contour list is divided into a number of buckets, each corresponding to one distance value, as suggested by Verwer (1989). This makes insertion and retrieval simple operations.

Figure 2 illustrates the handling of pixel pointers in the bucket structure. Each bucket is a set of pixel pointers, for example a linked list. The buckets are processed in order of increasing (distance) value. When a pixel pointer is fetched from a bucket, the pixel is inspected. If it is 1, then the pixel has already been reached by another part of the propagation process, and the pointer is discarded. Otherwise, the pixel is assigned to 1, and its neighbours are inspected. Pixel pointers to all neighbours that are 0 are put in appropriate buckets. These buckets will always have higher numbers (distance values) than the current bucket.

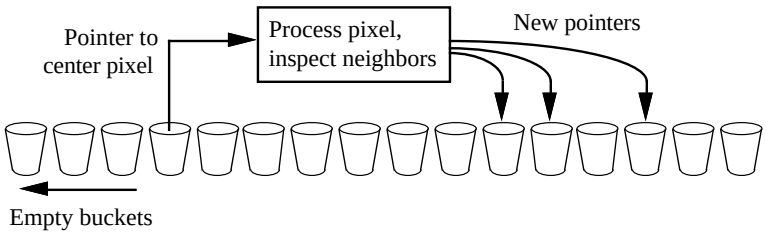


Figure 2. Bucket sorting applied to distance transformations. The buckets are processed in order of increasing distance values.

With integer values and bucket sorting, we no longer need any distance values stored in the contour list, since the distance value is given by the bucket in which the pointers are stored. Thus, the distance values are available *implicitly* from the bucket structure. For

Euclidean distance, though, a two-component vector must be put in the Contour List together with each pixel pointer.

Note that the bucket sorting approach can be applied to Euclidean distance, but preferably after a modification. To get integer indexes for the buckets, we let each bucket correspond to the square of some Euclidean distance value, calculated from the vectors as $v_x^2+v_y^2$. A much larger number of buckets is needed. For example, if we need to propagate up to the distance 128, $128^2 = 16384$ different distance values are possible. More generally speaking, in an n·n image, $(n-1)^2 \cdot 2$ different values are possible. Even though not all these values does actually occur as distance values (i.e. they can not be formed as the sum of two squares), there is no efficient way for our algorithm to exclude them. This implies that we would need a number of buckets of the same order as the image size, a memory cost of the same order as a distance map.

However, since we know that all pointers used are within a fixed distance (i.e. $\sqrt{2}$ pixel distances), we may reuse buckets. For example, between the distances 127 and 128, $128^2 - 127^2 = 255$ buckets are needed. When we store a pointer to a pixel assigned the distance d, it is stored in the bucket numbered (d^2 mod b), where b is the number of buckets available.

The entire algorithm runs in O(A) time. In Table 2 this algorithm is compared to the one described in Section 3 and the conventional raster scanning algorithm. The table is similar to Table 1, but the computation part is divided into initialization, distance transformation (DT) and thresholding.

Algorithm	Init.	DT	Thresholding	Memory
Scanning DT	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Limited CPDT	$O(n^2)$	O(A)	O(A)	$O(n^2)+O(l_{max})$
DM-free	$O(n^2)$	O(A)	O(A)	$O(l_{max})$

Table 2. Comparison of distance mapping based erosion or dilation algorithms.

The Euclidean version of the algorithm follows below as pseudo code. Non-Euclidean versions can be derived from this, but in such a case, the modulo operation is not as essential and could optionally be left out.

Dilation by Euclidean distance transformation without distance map,
pseudo code for the algorithm:
p is a pixel pointer (p_x, p_y)
b(p) is a pixel in the binary image b.
d is the propagation distance (structural element radius)
o is an offset vector (o_x, o_y).
v is a vector (v_x,v_y)
c is the current bucket number
B is the number of buckets available

(Note that all bucket entries hold both a pixel pointer and a vector.)

```
begin
for all p where b(p)=1 do
        if b(0,1)=0 or b(1,0)=0 or b(0,-1)=0 or b(-1,0)=0 then
                (p, (0,0)) is put in bucket 0
c := 0
while c < d do begin
        for all p,v in bucket c do
                if b(p) = 0 or v = 0 then begin
                        b(p) := 1
                        for all o ∈ {(0,1), (1,0), (0,-1), (-1,0), (0,1), (1,0), (0,-1), (-1,0)} do
                                if b(p+o)=0 then
                                        (p+o, v+o) is put in bucket (v_x+o_x)^2+(v_y+o_y)^2 mod B
                        end
        discard contents of bucket c
        c := c+1
        end
end.
```

Experimentally, the speed of the algorithm was close to the algorithm described in section 3.

5. Performing several consecutive erosions and dilations

The morphological operations *opening* and *closing* are defined below. Opening consists of an erosion followed by dilation. Closing consists of dilation followed by erosion. In this section we discuss the possibilities to optimize these operations for shorter execution time than what two independent erosions or dilations would demand.

Definition: We define opening and closing as:

Opening: $X_B = (X \ominus B) \oplus B$

Closing: $X^B = (X \oplus B) \ominus B$

Opening and closing are clustering operations. The operations are illustrated in figures 3-5. All erosions and dilations in the figures are computed with Euclidean metric.

Suppose we use the algorithm in section 4 for performing the erosions and dilations in the opening and closing operations. The erosion/dilation algorithm halts after processing the bucket with the distance corresponding to the erosion/dilation depth d. All buckets above this are normally ignored, but in this case we may reuse them to achieve an even faster algorithm.

We collect all the pixel pointers in these buckets, and put them all in bucket 0. These pixels are all pixels just beyond the contour. When this is done, we can start the second part of the opening or closing operation without scanning the image for edge pixels. Below is an outline of the opening algorithm.

Figure 3. Original image.

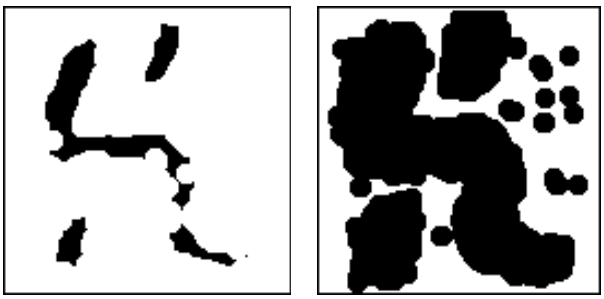


Figure 4. Eroded 5 steps and dilated 5 steps, respectively, Euclidean metric.

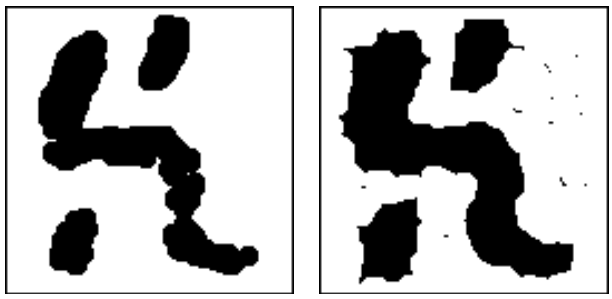


Figure 5. Left: Erosion 5 steps followed by dilation 5 steps. Right: Dilation 5 steps followed by erosion 5 steps. (Opening and closing, respectively.)

Algorithm Opening
1. Initialize contour list (bucket 0) by a full image scan.
2. Run DM-free erosion to distance d.
3. Initialize contour list by collecting pixels in unused buckets from step 2.
4. Run DM-free dilation to distance d.

The resulting algorithms can, from a computational complexity point of view, be considered one-pass opening and closing algorithms. They are fast algorithms that allow the Euclidean metric to be used in the erosions and dilations.

6. Conclusions

Thresholding of distance maps is an efficient method for performing erosion and dilation with Euclidean or approximately Euclidean metric. In this paper we have presented even faster algorithms based on generation of distance transforms by contour processing. Two new algorithms were suggested.

First, a contour processed distance transform may be halted after a specified number of iterations, just like a parallel algorithm. Therefore, the processing can be halted as soon as the area that is to be influenced by the erosion or dilation has been processed, and no other areas will be influenced. The algorithm will run in a time $\in O(A)$. The amount of memory needed will be $\in O(n^2 + l_{max})$.

The second method needs no the distance map, but is still a distance transformation algorithm. Like the previous algorithm, it halts after a number of iterations determined by the demanded erosion or dilation depth. The algorithm can use both Euclidean and non-Euclidean metric. In the non-Euclidean case, distance values are implicitly stored in the contour list, as the bucket number of the bucket that the pixel pointers are put in. The algorithm runs in a time $\in O(A)$ (not counting an initial scan for finding edge pixels), with a memory requirement is $\in O(l_{max})$ (not counting the binary source image).

A method for running this algorithm more than once without re-initialization was investigated. This method is useful for computing opening or closing.

References

Borgefors, B. (1986). “Distance Transformations in Digital Images”, *Computer Vision, Graphics and Image Processing* 34, pp 344-371.

Danielsson, P.E. (1980). “Euclidean Distance Mapping”, *Computer Graphics and Image Processing* 14, pp 227-248.

Dougherty, E.R, Giardina, C.R. (1987). *Image Processing - Continuous to Discrete, Vol. 1*, Prentice-Hall, Englewood Cliffs, New Jersey.

Laÿ, B. (1987). “Recursive Algorithms in Mathematical Morphology”, *Acta Stereologica, Vol. 6/III, Proc. 7th Int. Congress for Stereology*, Caen, France, pp 691-969.

Piper, J., Granum, E. (1987). “Computing Distance Transformations in Convex and Non-convex Domains”, *Pattern Recognition* 20, pp 599-615.

Ragnemalm, I. (1990). “Contour processing distance transforms”, *Progress in Image Analysis and Processing*, World Scientific, Singapore, pp 204-212.

Rosenfeld, A., Kak, A.C. (1976). *Digital Picture Processing*, Academic Press, New York.

Rosenfeld, A., Kak, A.C. (1982). *Digital Picture Processing*, Academic Press, New York.

Serra, J. (1982). *Image Analysis and Mathematical Morphology*, Academic Press, London.

Van Vliet, L.J., Verwer, B.J.H. (1988). “A contour processing method for fast binary operations”, *Pattern Recognition Letters* 7, pp 27-36.

Verwer, B.J.H, Verbeek, P.W, Dekker, S.T. (1989). “An Efficient Uniform Cost Algorithm Applied to Distance Transforms”, *IEEE Trans. on Pattern Analysis and Machine Intelligence* 11, pp 425-429.

Fast edge smoothing in binary images using Euclidean metric

In: Cantoni et. al., eds: Progress in Image Analysis II, World Scientific, Singapore, 1992.

Ge honom en slant eftersom han måste tjäna något på det han lär sig!

Give him threepence since he needs make gain by what he learns!

Euclid, said to a slave when a student asked what he would get by learning geometry

Fast edge smoothing in binary images using Euclidean metric

Ingemar Ragnemalm

Dept. of EE, Linköping University, S-581 83 Linköping, SwedenE-mail: ingemar@isy.liu.se

Abstract

Arcelli and Sanniti di Baja have developed an excellent method for smoothing jagged edges and suppressing salt-and-pepper noise from binary images. The method is more robust than methods using ordinary morphological operations. In this paper, we suggest a method for performing this algorithm in a both more time-efficient and memory-efficient way, using Euclidean metric and local operations.

Keywords: Distance transformation, propagation, contour processing, ordered propagation, binary image processing, Euclidean metric, edge smoothing.

1. Introduction

When an image has been made binary by thresholding, further analysis can be disturbed by the presence of noise. Typically, the noise tends to break up the edges of the objects in the image, or, in the case of salt-and-pepper noise, cause occasional pixels to be set in reset areas or reset in set areas. The noise will influence the topology of the objects in the image, thereby making analysis of the topology of objects difficult.

If the noise cannot be suppressed by other means, picture editing operations on the binary image can be needed. Such operations can be either context dependent or context independent. This paper will deal with general context independent methods that are not tailored for any particular application or situation.

The purpose is to remove small protrusions and dents from object borders as well as removing salt-and-pepper noise. One way to do this is to use a sequence of morphological operations, erosions and dilations. Serra (1982) treats the subject of morphological operations extensively. Fast algorithms for performing morphological operations have been suggested by Ragnemalm (1992) and Van Vliet and Verwer (1989).

Definition: An image consists of set pixels (objects) and reset pixels (background). The set pixels form the subset F = {1} and the reset pixels form the subset F' = {0}.

Definition: Opening of an image A is denoted O(A). Closing of the image A is denoted C(A).

Serra (1982) used the notation A_S and A^S for opening resp. closing of the image A by the structural element S. We have chosen a different notation since the structural element is not in focus here, but rather the sequence of operations.

Since the opening operation removes sparse set pixels as well as protrusions, while the closing operation removes sparse reset pixels and dents, applying both in sequence will often give an acceptable result. The two possibilities are opening of closing, O(C(A)) and closing of opening, C(O(A)). See also Serra (1982), page 418.

The operations O(C(A)) and C(O(A)) are not equivalent. If the image contains areas that are a more or less random mix of set and reset pixels, this entire area will be set by the opening of closing operation, while it will be reset by the closing of opening operation. This indicates that these operations are not so generally applicable and robust.

The problems are illustrated in Figures 1 and 2. If we try to smooth the first image in Figure 1 with C(O(A)), the result is the first image in Figure 2. If we try to smooth the first image in Figure 1 with O(C(A)), the result is the second image in Figure 2. As Figure 2 shows, none of them are edge smoothing operations, but rather clustering of white and black areas, respectively.

One might argue that the result of these operations are correct if the threshold for the thresholding operation that generated the binary image was rather high. For example, if pixels were classified as black only if that could be done with high confidence, then the clustering of black pixels can be justified. However, such an algorithm would not be context independent.

If the thresholding operation that classifies the pixels into black and white is unbiased, the result in Figure 1 (third image) is more desirable.

Arcelli and Sanniti di Baja (1988) developed a picture editing algorithm that solves this problem in a far more robust way, using distance transformations. It is outlined in section 3. The last image in Figure 1 shows the result of that algorithm. In section 4, we will suggest a faster version of this algorithm.

Parallel algorithms can be used for performing morphological operations like erosion, dilation, opening and closing. Such operations are very slow on general, single processor computers. However, they can be run in much less time by performing distance transformations followed by thresholding. Van Vliet and Verwer (1988) and Ragnemalm (1992) describe even faster algorithms for performing morphological operations.

2. Distance transformations

Distance transformations (DT) are useful tools for manipulation of binary images. The operation takes a binary image as input and generates an image where the value of each pixel tells the distance to the closest set pixel (feature pixel) of the binary image. The result is a *distance map*.

Many different distance metrics are possible. The simple City Block distance and Chessboard distance metrics were used by Rosenfeld (1966). The weighted Chamfer metrics, introduced by Montanari (1968) and refined by Borgefors (1986), are highly popular, since they give good approximations to the Euclidean distance for a low computational cost.

The best precision is achieved with *Euclidean distance transforms*, first described by Danielsson (1980). A Euclidean distance transform manipulates vectors rather than scalar distance values, and generates *Euclidean distance maps*. A *signed* Euclidean distance map uses vectors with signed components. This not only gives higher precision, but also the information about where the propagation originated.

Rosenfeld (1966) described the highly efficient sequential method for computing distance transforms. Danielsson's Euclidean algorithms were also sequential, but used a modified raster scanning pattern. Other versions of the sequential Euclidean distance transform have been suggested by Ye (1988) and Ragnemalm (1989).

More recently, contour processed (ordered propagation) distance transformations have been developed. Piper and Granum (1987) presented the basic method. Verwer et. al. (1989) introduced bucket sorting for better performance in the worst case. Ragnemalm (1990) presented algorithms for Euclidean distance. These algorithms propagate distance values in an ordered fashion. It is the kind of propagation method used in the algorithm presented in this paper.

3. Edge smoothing by raster-scanning DT

In this section we briefly describe the picture editing algorithm suggested by Arcelli and Sanniti di Baja (1988) as a reference for the new algorithm and for introducing symbols.

A DT is performed over both F and F'. The distance values are signed, with the sign indicating whether it is part of F or F'. We choose to let positive distance value denote F and negative denote F'.

Using a distance threshold d, we can now identify the pixel sets E_F and $E_{F'}$, where $E_F \subset F$ and $E_{F'} \subset F'$. E_F is the set of pixels within F with a distance to the closest pixel in F' lower than d. $E_{F'}$ is the set of pixels within F' with a distance to the closest pixel in F lower than d. We form the union of them as $E = E_F \cup E_{F'}$.

The value of the distance threshold d is tuned according to the desired resolution. A high threshold will identify larger areas as uncertain, thus belonging to the set E, and smooth larger dents and protrusions. The set E is the set of pixels that are considered uncertain.

When the set E is identified, all pixels in E are assigned the distance value = ∞. In practice, this means a distance value larger than the diagonal of the image.

Another DT is performed over E. The sign from the previous DT is kept, so each pixel in E is assigned the distance to the closest pixel in E', with the sign denoting if the originating (i.e. closest) pixel is in F or F'.

The resulting edited image A^+ consists of the sets $F^+ = \{1\}$ and $F^{+\prime} = \{0\}$. All pixels in the DT with positive sign are in F^+, while all pixels with negative sign are in $F^{+\prime}$. Thus, the image A^+ is generated by taking the sign bit from the last DT.

Figure 1 gives an example. The original image is an edge with an area where pixels are randomly set or reset. The middle image shows the set E' (that is, white pixels belong to E).

Figure 1. "Edge", original image, propagated areas, smoothed

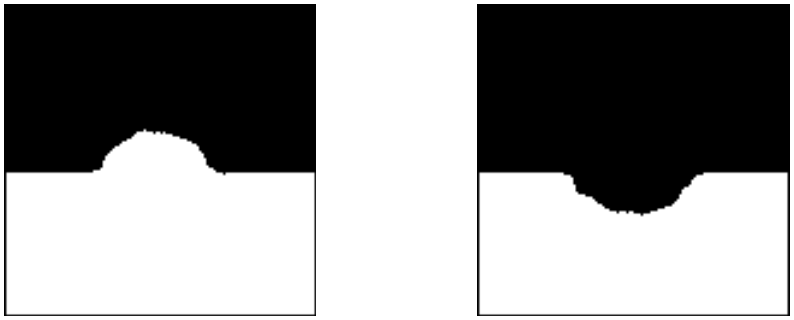


Figure 2. "Edge", closing of opening, opening of closing

It should be noted that while the original algorithm was designed using the chessboard distance metric, modifying it for more exact metrics is a trivial task. Using Euclidean distance transformations for the task is slightly less trivial. However, the new algorithm, described in the next section, does not only use Euclidean distance, but introduces some other techniques for optimizing the performance of the algorithm.

4. An algorithm using ordered propagation

The edge smoothing algorithm invented by Arcelli and Sanniti di Baja (1988), outlined in the previous section, can be combined with the techniques used by Ragnemalm (1992) in order to achieve a much faster algorithm, which also needs less memory.

The new algorithm uses the propagation method similar to the one that was used for performing erosion and dilation by Ragnemalm (1992). It is a contour processing (ordered propagation) method, where pointers to the current propagation front is stored in a list, the *Contour List*. A propagation method based on contour processing is described by Van Vliet and Verwer (1988), and distance mapping algorithms using the technique have been proposed by Piper and Granum (1987), Ragnemalm (1990) and Verwer et. al. (1989).

Following the approach of Verwer et. al. (1989), the Contour List is divided into a number of buckets, one for each possible distance value. For Euclidean metric, the squared distance value is used. See also Ragnemalm (1992).

The propagation method suggested by Ragnemalm (1992) was a DT algorithm that did not generate any distance map. Instead, the distance information was stored in the Contour List, and the successive erosion (or dilation) of the binary image involved provided the extra information needed for making the propagation correct and minimal. We will use the same technique here.

Note that we are using the Euclidean metric. To make this possible, we need to use vectors rather than distance values, as suggested by Danielsson (1980). This is actually an advantage for this particular application, since the vectors are pointing at corresponding originating pixels for the propagation. This information is needed in the last phase of the algorithm.

The sets F and F' are defined by the binary source image A. In this case, however, we need an extra binary image B. This is because we must be able to distinguish the sets F, F' and E. B is eroded in both directions from the edges in A. After the erosion is completed to distance d, the image holds E' (that is, E is the set of reset pixels in B).

Then, B is dilated from its edges. Suppose a pixel in position (x,y) in image B is reached by the dilation. Then we assign B(x,y) := 1. We also have the vector (u,v) available, which points to the originating pixel for the dilation to (x,y), so the originating pixel is in position (x+u, y+v). We assign $A^+(x,y) := A(x+u, y+v)$. This will give each pixel in A corresponding to a pixel in E the value of the closest pixel in E'. The resulting image A^+ can preferably be in the same memory space as A, so A will be overwritten by A^+.

The middle image in Figure 1 shows, as mentioned above, the set E'. That is, it shows the image B after the erosion process.

The algorithm is further speeded up by avoiding a full scan through B for edge detection. Instead, all unused pointers left in the Contour Set after the end of the erosion process can be collected into bucket 0 as an initialization of the dilation process. This method is described by Ragnemalm (1992), where it is used for opening and closing operations.

The algorithm is described in pseudo code below.

A[x,y] The binary value of the pixel on position (x,y) in the image A.
B[x,y] Ditto for the image B.
v_i The vector in the Contour List element i.
p_i Pointer to a pixel from the Contour List element i.
d The distance value chosen for propagation limit.

```
begin {start of main algorithm}

{scan A for edges}
for all pixels p do
    if a 4-neighbour n exists for which A[p]≠A[n] do
        Make(p, n-p)

{initialize B}
for all pixels p do B[p] := 1

{erosion of B}
for b := 0 to d² do
    for all elements i in bucket b do
        if B[pi] = 1 and A[pi + vi] = A[pi] then begin
            B[p] := 0
            for all 8-neighbours j of the pixel pi do
                if B[pj] = 1 then Make(pj , vi + pj - pi)
            end

{collect all old pointers}
for b := d² +1 to (d+1)² + 1 do
    for all elements i in bucket b do
        Make(i, (0,0))

{dilation of B, modify A}
b := 0
while the Contour List is not empty do begin
    for all elements i in bucket b do
        if B[pi] = 0 then begin
            B[pi] := 1
            A[pi] := A[pi + vi]
            for all 8-neighbours j of the pixel pi do
                if B[pj] = 0 then Make(pj , vi + pj - pi)
            end
    b := b + 1
    end
end {of main algorithm}

subroutine Make(p, v)
    allocate a new Contour List entry k.
    put k in bucket (vx² + vy²)
    vk := v
    pk := p
```

The following features have been left out from the pseudo code in order to simplify it. They are, however, important for the performance of the algorithm.

Since we have pointers to the originating pixel available at all stages in the propagation processes, it is possible to process only 2 or 3 neighbours for each pixel, as suggested by Ragnemalm (1990).

The number of buckets can be very large when using the squared distance. This unnecessary memory consumption can be avoided by using a modulo operation when accessing buckets, as suggested by Ragnemalm (1992).

The resulting algorithm scans the entire source image only once, to find the edges. All following computations are local, done only in the E set. Each pixel in the E set is visited twice, once in each propagation process. Since E is typically a rather small part of the image, we may consider the new algorithm to be essentially a one-pass algorithm. This should be compared to the 5 passes required for the algorithm by Arcelli and Sanniti di Baja (1988).

The memory demand is also lower than in the original algorithm. The original algorithm demand a greyscale image to store the distance transform. This is no extra expense in the case the binary image A is stored as a greyscale image, but it is if A is stored with 1 bit per pixel. Thus, for a source image of N·N pixels, we typically need N·N·8 bits of extra memory.

Our new algorithm requires enough memory to hold the Contour List plus 2 bits per pixel, original image included. The total extra memory is then N·N bits plus the Contour List. In a N·N image, the length of the edges are typically proportional to N, so the Contour List is usually very small. However, as mentioned by Van Vliet and Verwer (1988), the edges can be in O(N·N) in the worst case. (The O set should not be confused with the operation O, used in section 2.)

Finally, we give one more computer generated example, in Figure 3. The figure shows an image with a wider collection of artifacts, including protrusions, dents, salt-and-pepper noise, thin cracks and thin connections. The example uses a bigger structural element (propagation distance) than the previous example.

Figure 3. A more general smoothing example.

5. Conclusions

We have presented a fast version of the smoothing algorithm by Arcelli and Sanniti di Baja (1988), which is an algorithm for performing context independent smoothing of binary images. As shown in our examples, this algorithm give much better result than methods using consecutive erosions and dilations, in the sense that it does enlarge or shrink the areas processed, but rather cut uncertain areas in half between the white and black sets.

The algorithm is computed after a single scan of the image (for detecting edge pixels), followed by local operations. Thus, the algorithm runs in very short time in the typical case, since it only needs to process a small part of the image as soon as the initial, very fast scan is completed.

The memory demand is low. It needs enough memory to hold the Contour List, which is typically of a size in O(N) (if the image to be smoothed is a N·N image). It also needs an extra binary image of size N·N. This should be compared to the demand of the full greyscale distance map that the original algorithm needs, demanding 8 bits or more per pixel.

References

C. Arcelli, G. Sanniti di Baja (1988), "Picture Editing by Simultaneously Smoothing Figure Protrusions and Dents", *Proceedings of the 9th International Conference on Pattern Recognition,* Rome, pp 948-950.

G. Borgefors (1986), "Distance Transformations in Digital Images", *Computer Vision, Graphics and Image Processing* 34, 344-371.

P.E. Danielsson (1980), "Euclidean Distance Mapping", *Computer Graphics and Image Processing* 14, 227-248.

U. Montanari (1968), "A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance", *Journal of the ACM* 15, 600-624.

J. Piper, E. Granum (1987), "Computing Distance Transformations in Convex and Non-convex Domains", *Pattern Recognition* 20, 599-615.

I. Ragnemalm (1989), "The Euclidean Distance Transform and its implementation on SIMD architectures", *Proceedings, 6th Scandinavian Conf. on Image Analysis,* Oulo, 379-384.

I. Ragnemalm (1990), "Contour processing distance transforms", in: Cantoni et. al. ed, *Progress in Image Analysis and Processing,* World Scientific, Singapore, 204-212.

I. Ragnemalm (1992), "Fast erosion and dilation by contour processing and thresholding of Euclidean distance maps", *Pattern Recognition Letters* 13, 161-166.

A. Rosenfeld (1966), J.L. Pfaltz, "Sequential Operations in Digital Picture Processing", *Journal of the ACM* 13, 471-494.

J. Serra (1982), *Image Analysis and Mathematical Morphology,* Academic Press, London.

L.J. Van Vliet, B.J.H. Verwer (1988), "A contour processing method for fast binary operations", *Pattern Recognition Letters* 7, 27-36.

B.J.H. Verwer, P.W. Verbeek and S.T. Dekker (1989), "An Efficient Uniform Cost Algorithm Applied to Distance Transforms", *IEEE Trans. on Pattern Analysis and Machine Intelligence* 11, 425-429.

Q.Z. Ye (1988), "The Signed Euclidean Distance Transform and Its Applications", *Proceedings, 9:th International Conference on Pattern Recognition,* Rome, 495-499.

The Euclidean Distance Transform: Finding the Local Maxima and Reconstructing the Shape

Proceedings, 7th Scandinavian Conference on Image Analysis, Aalborg, 1991.

The Euclidean Distance Transform: Finding the Local Maxima and Reconstructing the Shape

Gunilla Borgefors[1] Ingemar Ragnemalm[2] Gabriella Sanniti di Baja[3]

1) Swedish Defence Research Estab., Box 1165, S-581 11 Linköping, Sweden2) Dept. of EE, University of Linköping, S-581 83 Linköping, Sweden3) Istituto di Cibernetica C.N.R., Via Toiano 6, I-800 72 Arco Felice, Italy

Abstract

The Euclidean distance transform measures distances in digital images exactly. However, due to the discrete grid, difficulties occur when using the Euclidean distance that do not occur when using simpler – and less accurate – distance transforms (e.g. city block and chessboard). In this paper we discuss the problem of characterizing and extracting the local maxima in the Euclidean distance transform. The local maxima are necessary and sufficient to exactly reconstruct the original shape. They do not usually form a connected set, and are thus only a subset of the skeleton (or medial axis) of the shape. An algorithm for extracting the local maxima is presented. Reconstruction of the shape from the local maxima is achieved using a reverse Euclidean distance transform. The resulting shape is exactly the original one, but the distance values are different from the original distance transform. This new type of Euclidean distance transform promises to be useful in some applications.

1. Introduction

One of the most important applications for distance transformations of binary images is skeletonization, that is to extract a description of the shape using a small number of pixels. Two different types of structures are commonly used.

The first type of structure is obtained by extraction of *local maxima* from a distance map. This returns a pixel set which, along with the corresponding distance values from the distance map gives a complete description of the shape(s). Rosenfeld introduced this structure and used the name *distance skeleton* for the set of local maxima [12]. Many papers have been written on distance skeletons for different distance metrics. Danielsson described algorithms for the Euclidean distance metric [4].

The set of local maxima is usually not connected. To get a connected set, that is a connected skeleton, one has to add a number of linking pixels, using rules that are dependent on the distance transform used. The original shape can be recovered by merging a number of digital disks, one for each pixel in the set of local maxima, where the radius of each disk is the distance value of its central pixel. As it is impossible to distinguish between the local maxima and the linking pixels in the computed skeleton, the disks will also be applied to the linking pixels. These extra disks will be completely included in the set of disks from the local maxima.

The other type of structure is the skeleton (or medial axis) that is the result of *thinning* operations. Thinning operations are classically performed with parallel neighbourhood operations, that deletes edge points in a connectivity preserving fashion [5]. Hence, this operation is topology-preserving. There is, however, usually no possibility of reconstruction. For this type of skeleton also, many different algorithms have been suggested. A special case of thinned images is produced by inspecting the Voronoi edges in a Euclidean distance map, as suggested by several authors [6, 8]. An even more ingenious method, using “snakes”, is suggested by Leymarie et. al. [7].

In this paper, we will discuss the first of these two data structures. We will limit the discussion to finding local maxima, and will not try to link them. These discussions are closely related to those in [4], but go deeper and give further details. We also propose a slightly different algorithm. The reconstruction of the shapes can be done in several ways. We propose a new reverse Euclidean distance transform. The resulting shape is exactly the original one (if an error free Euclidean distance transform is used), but the distance values are different from the original distance values. This might be an advantage rather than a drawback, as it provides a further characterization of the shape.

2. Distance maps

A distance transformation (DT) is an operation that generates a distance map (DM) from a binary image. In the DM, each feature (object) pixel gets the distance to the closest non-feature (background) pixel of the binary image.

On all but massively parallel machines, DMs in general are preferably generated using sequential operations, as described already in [12], or with contour processing techniques [9, 10, 13]. In these cases, the computational complexity is linearly dependent on image size, which is the optimal case. (Borgefors et. al. has described an algorithm for massively parallel machines, also with linear complexity [2].)

Euclidean distance maps (EDM) and Euclidean distance transformations (EDT) were first described by Danielsson [4]. In an EDM, each pixel does not hold just a single distance value, but rather a two-component vector. If the EDM is signed, [15], the vector points to the closest background pixel.

Most sequential EDTs suggested are not totally error-free, though the errors are very small. The one from [4] uses two major scans over the image, where each scan goes forwards and backwards over each row. The small errors can give incorrect reconstruction of the original shape, where the difference is a single edge pixel. The problem is unlikely to have any major impact on practical applications. However, in this paper we assume that we use an error-free EDT, like those suggested in [10, 14].

3. Extraction of local maxima

Rosenfeld et. al. [12] uses a straight-forward algorithm for detection of local maxima. This is possible for "simple" metrics like the city block distance and chessboard distance, but it is not applicable to the Euclidean distance. In fact, it is not applicable to the weighted distance transforms either. There the local distances between horizontal/vertical and between diagonal neighbours are different, [2]. (A popular weighted DT uses the values 3 and 4 respectively.) Arcelli and Sanniti di Baja has characterized local maxima in weighted DMs, [1].

In this section, we will describe how the local maxima can be extracted from an EDM. In order to do this, we must first discuss the concept of digital disks.

We use the following definitions:

The *distance between two square pixels* is the Euclidean distance between their centres.

A *circular disk* (cdisk) is a continuous circular disk with some radius **d**, centered around a point with integer coordinates (and therefore centered around the pixel in that location).

A pixel is *inside* a cdisk with radius **d** if and only if its centre is at a distance less than **d** from the centre of the disk. Thus, a pixel can be inside a cdisk even if parts of the pixel reach outside the cdisks perimeter. Expressed in another way, a cdisk *encloses* a pixel if and only if the pixel is inside the cdisk.

A *digital disk* (ddisk) is the set of pixels inside a circular disk with radius **d**, that is, the set of pixels with centres on a distance < **d** from the central pixel.

These definitions are exemplified by Figure 1, which also hints the difficulties introduced by the discrete grid. Each pixel in an EDM holds a vector with a length equal to **d**, where **d** is the distance to the closest background pixel (centre to centre). In other words, for each pixel in an EDM, a cdisk with radius **d** centered in the pixel will only enclose object pixels, and will have one or more background pixels centered on its perimeter, exactly at the distance **d**.

The length of a vector (x,y) is $\mathbf{d} = \sqrt{x^2 + y^2}$. Hence, the squared length $\mathbf{d}^2$ of a vector in an EDM is always an integer, since the vector components are integers.

The strict definition of *local maxima* is the minimal set of pixels where the union of all the corresponding cdisks enclose all object pixels, but no background pixels. (In practice it is difficult to guarantee that the set of detected local maxima is minimal.)

The set of points enclosed by a circular disk, the ddisk corresponding to some cdisk, is the same for a range of cdisk radii. Hence, for each cdisk, there exists only one ddisk, but for each ddisk there exist several corresponding cdisks. This is true even if we only consider distance values that are square roots of integers. However, all cdisks from an EDM has one or more background pixels on the perimeter of the disk. This means that if we limit the set of possible cdisks to the set that can correspond to distance values in

an EDM (that is, with radii that squared are the sums of the squares of two integers), we have a one-to-one mapping between cdisks and ddisks.

Consider two pixels **a** and **b**, with associated cdisks with radii $d_a > d_b$ centered around **a** and **b** respectively. If the cdisk of **a** encloses all pixels enclosed by **b**, then **b** is *not* a local maximum. Note that this does *not* imply that the cdisk of **a** must completely cover the cdisk of **b**. The ddisk of **a**, however, completely covers the ddisk of **b**.

We can now present our algorithm for extracting local maxima. For each object pixel, the eight neighbouring pixels are inspected to check if any of them define a disk that completely covers the ddisk of the centre pixel. However, it *is* possible that none of the eight neighbours do so, but that nevertheless a pixel farther away does define such a disk. In that case, we will include one extra pixel in the set of local maxima. These cases correspond to the rare cases where the EDT as suggested by Danielsson produce errors [4].

The extra pixels will of course not change the result of the reconstruction algorithm, as the pixels they can add to the shape are also included in other disks. In fact, they will make the set of local maxima more connected than it otherwise would be. The only problem that could

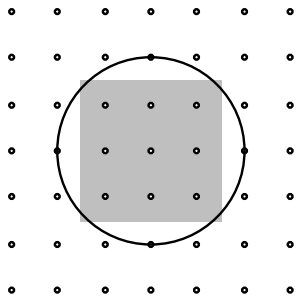


Figure 1. A circular disk (cdisk) with radius 2. The corresponding digital disk (ddisk) consists of the nine pixels enclosed by the cdisk (shadowed area).

occur is that the end points of skeletal branches might not be correctly identified. An end point should preferably be a true local maximum. However, as stated above, these points are very rare.

Now, we are ready to formulate our definition, used for extracting local maxima:
Definition: A pixel is a *local maximum* in the EDM if the set of points enclosed by its associated disk is not completely covered by any of the disks defined by any of the pixel's 8-neighbours.

Danielsson [4], in fact, suggested this rule, but formulated it differently. The rule will not produce a minimal set of locally maximal points, even if we disregard the extra pixels introduced by not checking pixels farther away than the 8-neighbours. If a ddisk is not covered by any other single ddisk, but can be covered by the *union* of two or more other ddisks, the corresponding pixel could be removed from the set of local maxima. This case, however, is not detectable in any efficient way.

In order to implement this rule, we need look-up tables that determine when one disk covers the disk of a neighbour. For each distance value that can occur in an EDM, consider a disk with that radius. The table should give the radius of the smallest disk that

covers that disk, centered either on a horizontal or vertical neighbour or centered on a diagonal neighbour. We refer to these radii as *hlut* and *dlut*, respectively. Table 1 gives the look-up table for small radii. The principle is illustrated in Figure 2.

Note the many blank entries in the hlut and dlut tables. They are for radii that can not occur in an EDM. These blank entries do not occur in any regular order, but with increasing frequency at higher radii. Because of the lack of order in which these blank entries occur, we can not easily make the look-up tables smaller by skipping them.

Table 1. Look-up tables hlut *and* dlut *for extraction of local maxima, for disks (that is EDT values) with squared radii up to 80.*

d^2	hlut	dlut	d^2	hlut	dlut	d^2	hlut	dlut	d^2	hlut	dlut	d^2	hlut	dlut
1	2	4	18	29	32	35	-	-	52	68	73	69	--	-
2	5	8	19	-	-	36	49	53	53	68	80	70	--	-
3	-	-	20	29	34	37	50	53	54	-	-	71	-	-
4	8	9	21	-	-	38	-	-	55	-	-	72	89	97
5	10	13	22	-	-	39	-	-	56	-	-	73	89	100
6	-	-	23	-	-	40	52	58	57	-	-	74	97	100
7	-	-	24	-	-	41	58	61	58	72	80	75	-	-
8	13	16	25	32	36	42	-	-	59	-	-	76	-	-
9	16	20	26	37	45	43	-	-	60	-	-	77	-	-
10	17	20	27	-	-	44	-	-	61	74	81	78	-	-
11	-	-	28	-	-	45	58	64	62	-	-	79	-	-
12	-	-	29	40	45	46	-	-	63	-	-	80	97	101
13	18	25	30	-	-	47	-	-	64	80	89			
14	-	-	31	-	-	48	-	-	65	82	89			
15	-	-	32	41	49	49	61	68	66	-	-			
16	25	26	33	-	-	50	65	68	67	-	-			
17	26	29	34	45	52	51	-	-	68	85	90			

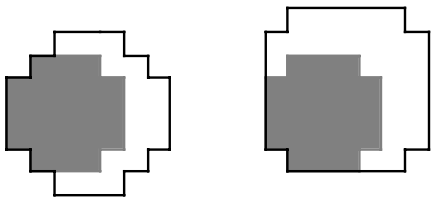


Figure 2. Illustration of an entry in Table 1. For a disk with squared radius 8 (shaded), we get hlut[8]=13 and dlut[8]=16. A disk centered on a horizontal neighbour (left) must have a squared radius ≥ 13 to cover the smaller disk. For a diagonal neighbour (right), the number is 16.

Given these look-up tables, the local maximum extraction algorithm is very simple. Figure 3 shows the result on an example image.

This extraction method is basically the same as the SKED algorithm suggested by Danielsson [4]. Our version uses a different kind of look-up table. Danielsson uses a look-up table indexed by the two vector coordinates, and four vectors per index, while we index with the squared distance value and have two integers per index. In both cases, we need a look-up table of the size $O(N^2)$ for an N^2 image.

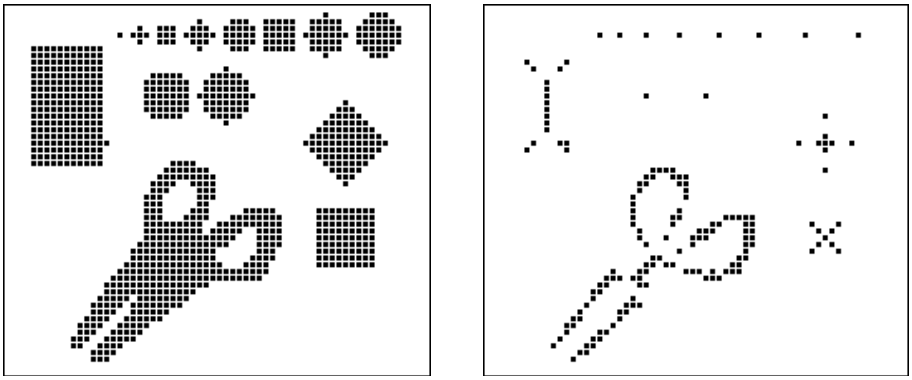


Figure 3. Result of extraction of local maxima.

4. Reconstruction by reverse distance transformation

In this section, we will describe methods for reconstructing the original binary shapes from the local maxima, or, indeed, from a connected distance skeleton (if such a structure have been computed) .

The first algorithm suggested for solving this problem was Danielsson's SKED^{-1} [4]. The algorithm uses the same look-up table as the SKED algorithm (mentioned in the previous section). The SKED^{-1} does not just reconstruct the binary shapes, but the entire EDM.

Now, we suggest an alternative algorithm for reconstructing the original shapes. It does not reconstruct the original EDM. Instead, we generate a distance map where all vectors point to local maxima in the EDT rather than to the background, that is, the direction of the vectors is reversed compared to the original EDM. Because of this, we call the algorithm a *reversed* EDT. The distance values tell the distance to the perimeter of the cdisks rather than to background pixels.

We name the algorithm Reverse Euclidean Distance Transform (REDT). It scans the image similarly to an EDT. Here we use scans identical to the 3-scan EDT suggested by Ragnemalm [11]. See Figure 4.

For each pixel, some of the eight neighbours are inspected at each scan. If any neighbour is a local maximum, a vector from the centre pixel to that neighbour is constructed, and the length of that vector is subtracted from the distance in the neighbour. This difference is a candidate distance for the pixel being processed. If the neighbour is not a local maximum, but holds a vector pointing to such a pixel, a vector to that local maximum is constructed and subtracted from the distance value in it. This difference is another candidate value for the centre pixel. If the processed pixel itself already holds a vector to a local maximum, the sum of that vector and the distance value in that local maximum is added and also sums to a candidate value.

The centre pixel then is assigned the highest of the candidate values, along with the associated vector. If the neighbour is neither a local maximum nor holds a vector to a local maximum, it will not generate any candidate value.

The REDT algorithm in pseudo-code becomes:

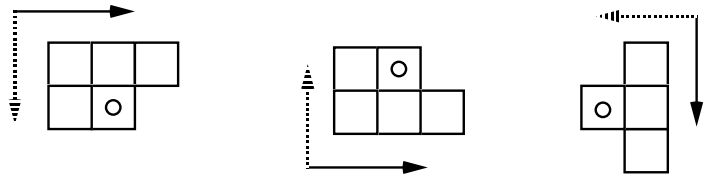


Figure 4. Neighbourhoods for a 3-scan EDT. They are also used for the REDT algorithm.

Algorithm REDT
d_c and **v_c** are the candidate distance value and corresponding vector
d_t and **v_t** are temporary variables for possible new d_c and v_c
V(p) is the vector in the pixel p
D(p) is the distance value in the pixel p
L(v) is the length of the vector v

```
for y:=1 to N do for x:=1 to N
    for all p := (x+i,y+j), (i,j) ∈ {(0,0) (1,-1) (0,-1) (-1,-1) (-1,0)}  Check_Candidate(p)
        V(x,y) := v_c
        D(x,y) := d_c
for y:=N to 1 do for x:=1 to N
    for all p := (x+i,y+j), (i,j) ∈ {(0,0) (-1,0) (-1,1) (0,1) (1,1)}  Check_Candidate(p)
        V(x,y) := v_c
        D(x,y) := d_c
for x:=N to 1 do for y:=1 to N
    for all p := (x+i,y+j), (i,j) ∈ {(0,0) (1,1) (1,0) (1,-1)}  Check_Candidate(p)
        V(x,y) := v_c
        D(x,y) := d_c

procedure Check_Candidate(p)
    if p belongs to the set of local maxima
        v_t := (i,j)
        d_t := D(p) - L(i,j)
        if d_t > d_c then d_c := d_t; v_c := v_t
    else if p+V(p) belongs to the set of local maxima
        v_t := V(p) + (i,j)
        d_t := D(p + V(p)) - L(v_t)
        if d_t > d_c then d_c := d_t; v_c := v_t
```

Finally, we will briefly mention some practical aspects on the implementation of the REDT algorithm, that are not obvious from the pseudo-code above.

First, since there is no way to use only integer numbers while maintaining the precision achieved by using vectors, we use fixed point numbers in the experimental implementation. This is fast and practical for the problem, since we know the range of possible numbers well. We don't need the flexibility of floating point numbers.

Second, to avoid computing the square root for finding the length of a vector, we use a look-up table of vector lengths. It is indexed with the two components of a vector, and gives the length as a fixed point number. This look-up table must have the size N/2 by N/2 for N·N images, if the shapes can be contained inside the image.

5. Conclusions

We have discussed the extraction of local maxima from the Euclidean distance map, leading to the suggested extraction method, which is simple once appropriate look-up tables have been constructed.

For the reconstruction problem, we have described a new method, which generates a kind of distance map that provides other information than the normal Euclidean distance map does, since the vectors point to the local maxima (or connected skeleton) rather than the edge of the object. The distances produced by this algorithm refer to the distance to the edges of the continuous disks, disks resulting from the local maxima extraction procedure.

Correct reconstruction is ensured if an error-free EDT is used. The original EDT algorithms [4] does not produce completely error-free distance maps, which may cause the reconstruction to include a few extra pixels at the border in some rare cases.

We have not yet constructed a connected Euclidean skeleton. Some linking pixels must then be added to the set of local maxima. However, for data compression and reconstruction purposes the local maxima are sufficient.

References

[1] C. Arcelli, G. Sanniti di Baja, “Weighted Distance Transforms: A characterization”, *Image Analysis and Processing II,* Plenum Press 1988, pp 205-211.

[2] G. Borgefors, T. Hartmann, S.L. Tanimoto, “Parallel Distance Transforms on pyramid machines: Theory and Implementation”, *Signal Processing* 21, 1990, pp 61-86.

[3] G. Borgefors, “Distance Transformations in Digital Images”, *Computer Vision, Graphics and Image Processing* 34, 1986, pp 344-371.

[4] P.E. Danielsson, “Euclidean Distance Mapping”, *Computer Graphics and Image Processing* 14, 1980, pp 227-248.

[5] C.J. Hilditch, “Linear Skeletons from Square Cupboards”, *Machine Intelligence 4*, Univ. Press, Edinburgh, 1969, pp 403-420.

[6] F. Klein, “Euclidean Skeletons”, *Proc. 5th Scandinavian Conf. on Image Analysis*, Stockholm, 1987, pp 443-450.

[7] F. Leymarie, M.D. Levine, “Skeletons from Snakes”, *Progress in Image Analysis and Processing,* World Scientific, Singapore, 1990, pp 186-193.

[8] R. Ogniewicz, M. Ilg, “Skeletons with Euclidean metric and correct topology and their application in Object Recognition and Document Analysis”, *Proc. 4th Int. Symposium on Spatial Data Handling*, Zürich, 1990, pp 15-24.

[9] J. Piper, E. Granum, “Computing Distance Transformations in Convex and Non-convex Domains”, *Pattern Recognition* 20, 1987, pp 599-615.

[10] I. Ragnemalm, “Contour processing distance transforms”, *Progress in Image Analysis and Processing,* World Scientific, Singapore, 1990, pp 204-212.

[11] I. Ragnemalm, “The Euclidean Distance Transform and its implementation on SIMD architectures”, *Proc. 6th Scandinavian Conf. on Image Analysis*, Oulu, 1989, pp 379-384.

[12] A. Rosenfeld, J.L. Pfaltz, “Sequential Operations in Digital Picture Processing”, *Journal of the ACM* 13, 1966, pp 471-494.

[13] B.J.H. Verwer, “Improved Metrics in Image Processing applied to the Hilditch Skeleton”, *Proc. 9:th International Conf. on Pattern Recognition*, Rome, 1988, pp 137-142.

[14] H. Yamada, “Complete Euclidean Distance Transformation By Parallel Operation”, *Proc. 7:th International Conference on Pattern Recognition*, Montreal, 1984, pp 69-71.

[15] Q.Z. Ye, “The Signed Euclidean Distance Transform and Its Applications”, *Proc. 9:th International Conference on Pattern Recognition*, Rome, 1988, pp 495-499.

The Euclidean Distance Transform in arbitrary dimensions

Proceedings, Int. Conf. on Image Processing and its Applications, Maastricht, 1992.
Accepted for publication in Pattern Recognition Letters.

Den kortaste vägen mellan två punkter är en rät linjal.

(Not quite as corny in english:) The shortest path between two points is a straight ruler.

M. Ljung, “Kromosomtalaren”

The Euclidean Distance Transform in arbitrary dimensions

Ingemar Ragnemalm

Image Processing Group, Dept. of EE, Linköping University,S-581 83 Linköping, Sweden E-mail: ingemar@isy.liu.se

Abstract

The original sequential Euclidean Distance Transformation is not separable. This makes it useful only on single processor systems. We suggest variants for 2, 3 and arbitrary dimensions that are separable, suitable for various parallel architectures. The results include a 4-scan algorithm for 3-dimensional images.

Keywords: Euclidean metric, distance transformation, arbitrary dimensions, raster scanning.

1. Introduction

The Euclidean Distance Transform (EDT), invented by Danielsson (1980), allows the generation of distance maps with no significant errors. Later versions of the algorithm, by Yamada (1984) and Ragnemalm (1990), generate completely error-free Euclidean distance maps. The superior precision of the EDT over other distance transformation algorithms is possible due to the use of vectors instead of scalar values for the propagation of distance values. Pseudo-Euclidean algorithms, suggested by Montanari (1968) and Borgefors (1986) can get close to Euclidean metric, but they always have errors proportional to distance.

The generalization of distance maps, and Euclidean distance maps in particular, to 3 or arbitrary dimensions has received relatively little attention. Algorithms for 3D with some generalization have been described by Mohr and Bajcsy (1983) and Borgefors (1984).

In this paper, a number of new algorithms for EDT in 3 and higher dimensions are presented. We will not describe the processing of EDT in detail, but rather define algorithms with the neighbourhoods (masks) used. The full algorithms for Euclidean distance transformation are described in more detail by Danielsson (1980) and Ye (1988). We limit the discussion to raster scanning, sequential algorithms, as proposed by Rosenfeld (1966).

2. Separable Euclidean distance transforms

We define a *separable* EDT algorithm as an algorithm where the scans can be applied independently, in any order. A *non-separable* EDT algorithm uses scans that are dependent of each other, combining several masks in one scan. The terms separable and non-separable are equivalent to the terms *single-scanning* resp. *double-scanning*, as used by Ragnemalm (1989).

In the original 8SSED algorithm, as proposed by Danielsson (1980) and illustrated in Figure 1, each line is scanned back and forth, thereby making two scans of each line for each scan through the image. Hence, this algorithm is not separable. The same holds for the algorithms in 3 and higher dimensions by Mohr and Bajcsy (1983) and Borgefors (1984).

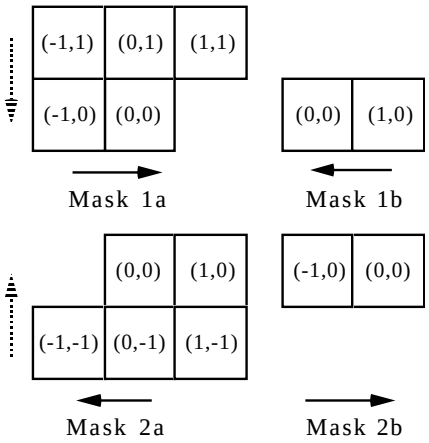


Figure 1. The original 2-dimensional EDT.

In Figure 1, each mask pair 1a, 1b resp. 2a, 2b result in propagation in half of all possible directions. Each pixel holds a two-component vector. These vectors, pointing to the center pixel (0,0), are the offset vectors used in the algorithm. These vectors will be omitted in the following figures.

One separable Euclidean distance transform for 2-dimensional images is the three-scan EDT suggested by Ragnemalm (1989) and illustrated in Figure 2.

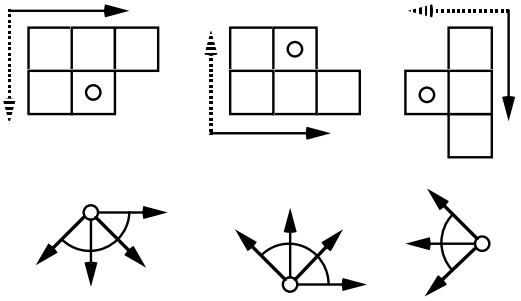


Figure 2. 3-scan EDT.

Separable algorithms are useful for implementation on parallel architectures, but they are also quite useful on single processor systems. In this paper, we will present separable algorithms for 3 and arbitrary dimensions. We will put the following constraints on our algorithms.

• We use only immediate neighbours, that is, for the voxel in position (x_1,x_2,x_3…) we may use neighbours at positions (x_1+d_1, x_2+d_2, x_3+d_3…) where $d_i \in$ { -1, 0, +1 }.

• We use separable algorithms, since they can be implemented in parallel.

• We try to minimize the number of scans.

3. The direction space

In a separable Euclidean distance transform, each scan will support propagation over some part of the direction space, and the union of these parts should cover the entire direction space completely. This necessary requirement is discussed by Ragnemalm (1989) for the 2-dimensional case.

In 2-dimensional images, the direction space is simple; a 1-dimensional unit circle. It is very simple to analyze algorithms to see what parts of the direction space are supported by each scan, and then confirm that the entire direction space is covered. Such a coverage check is easily done e.g. in Figure 2 above.

The direction space for 3-dimensional images is a 2-dimensional space, a unit sphere. In order to analyze and visualize this space, a straightforward approach could be to map the directions on this sphere. However, a segmented sphere is hard to represent on a 2D plane.

Instead, we will use a different method which we call *Unfolded Cube Graphs.* In this case, the direction space is mapped on a cube, which we can display unfolded in 2D. This kind of graph is more suitable than graphs using spheres, not only because it is easier to map on a 2D plane, but also because of the close relation between the Cartesian grid and the direction segments to be supported.

Figure 3 illustrates how the Unfolded Cube Graph works. The needle in the figure defines a direction from the center of the cube. This direction is mapped on a point in the Unfolded Cube Graph.

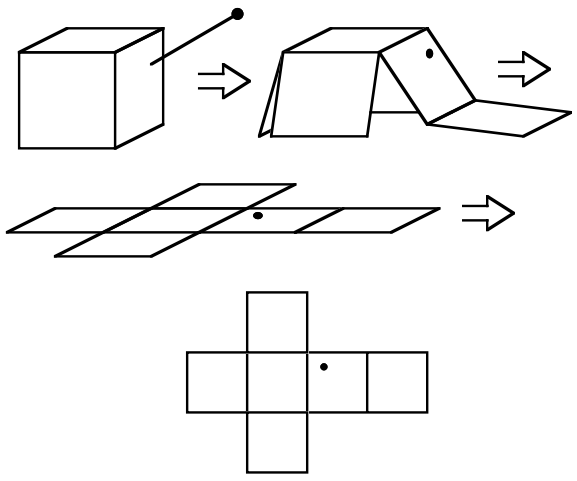


Figure 3. The Unfolded Cube Graph.

4. From direction space to algorithms

With the Unfolded Cube graph as a tool, we will now search for 3D operators useful for a 3D version of EDT. As previously stated, we are only interested in algorithms using neighbourhoods which are subsets within the 3·3·3 neighbourhood. It is of no use to include voxels forward in the scanning directions since they do not contribute to the directions supported. Hence, the largest reasonable mask within 3·3·3 is the one shown in Figure 4.

It is not difficult to find that the Unfolded Cube Graph for such a mask, displaying the parts of the direction space where propagation is supported by the mask, is the one shown in Figure 5. This Unfolded Cube Graph, as well as the following ones, can be found directly from the mask, but have also been verified with computer experiments.

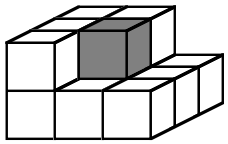


Figure 4. The largest mask within a 3·3·3 neighbourhood useful for separable EDT algorithms.

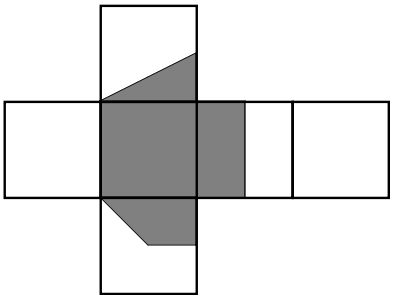


Figure 5. The Unfolded Cube Graph corresponding to the mask in Figure 4.

The fewer scans we need, the faster the algorithm will run. By inspecting the Unfolded Cube Graphs, we have found a solution using only four scans, illustrated in Figure 6. The four scans are symmetrical. The Unfolded Cube Graph for one scan is shown in Figure 7.

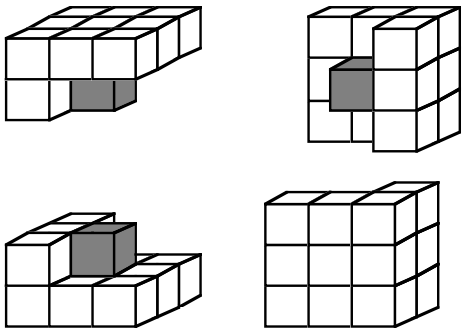


Figure 6. The four masks in the 4-scan algorithm, the algorithm with the smallest possible number of scans.

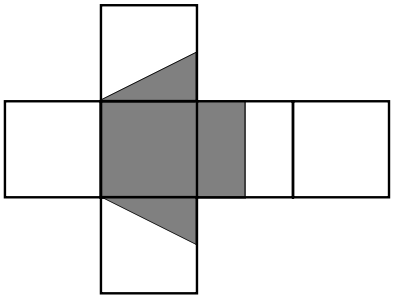


Figure 7. The corresponding Unfolded Cube Graph for one of the masks in Figure 6. (Bottom left.)

While it is possible to make mask sets of other shapes, it is not possible to make a 3D algorithm of this kind with fewer than 4 scans. We will give a brief outline of a proof of this statement.

Consider an arbitrarily small circle in the center of each of the six sides of the cube in the Unfolded Cube Graph. All circles (as well as the rest of the direction cube) must be covered by at least one of the scans used. See Figure 8.

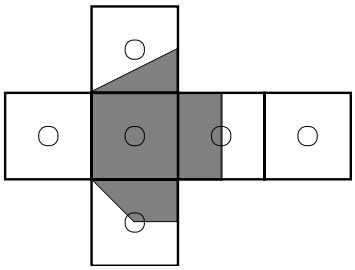


Figure 8. The centres of each side of the cube.

In Figure 8, we can see that one single scan like the one in Figure 4 will cover 1 + 1/2 + 3/8 = 1 7/8 circles out of the six circles. Hence, it is impossible to fill the entire Unfolded Cube Graph with only three scans.

We claim that larger neighbourhoods than 3·3·3 are not of practical interest. To motivate this claim, we examine the 5·5·5 case. The largest neighbourhood is shown in Figure 9. The mask demands significantly more processing time compared to the mask in Figure 4, but the gain in direction space is marginal. See Figure 10. In particular, the proof above still holds for this mask, so it is still impossible to make an algorithm with less than 4 scans.

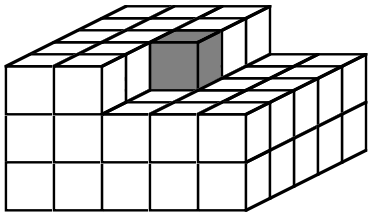


Figure 9. The largest useful mask within a 5·5·5 neighbourhood.

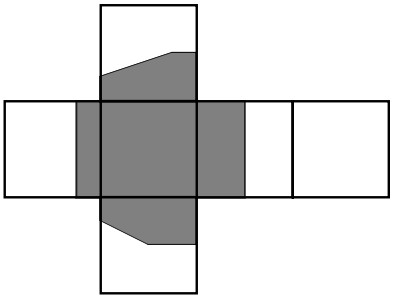


Figure 10. The Unfolded Cube Graph corresponding to the mask in Figure 9.

The 4-scan algorithm is related to the 3-scan algorithm in 2 dimensions [7]. Apparently, these optimal cases can not be generalized into arbitrary dimensions. For each space, we can find an optimal case. In this paper, we will not try finding it for higher dimensions than 3D.

5. Arbitrary dimensions

The optimal solutions for 2D and 3D (3-scan and 4-scan, respectively) have proven difficult to generalize to higher dimensions. However, we have found sub-optimal algorithms for which this is possible.

While pixels in 3D are generally called *voxels*, and Borgefors (1984) suggests the name *rexel* for a pixel in 4-dimensional space, there appears to exist no name in the literature for pixels in arbitrary dimensions. We suggest the name *hoxel* (high order pixel).

We will describe one algorithm which is easy to generalize due to symmetry, and which can be modified for various neighbourhood sizes. We call it the *Corner EDT*. Though this is far from the optimal solution, it is reasonably fast. Its simplest 3D version consists of eight masks, each with the shape shown in Figure 11.

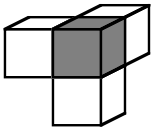


Figure 11. One of eight masks needed for 6-neighbour Corner EDT.

This is the minimal, 6-voxel neighbourhood version of the Corner EDT. It is easy to modify it to 18- or 26-voxel neighbourhoods. In all these cases, the Unfolded Cube Graph will be identical. It is shown in Figure 12.

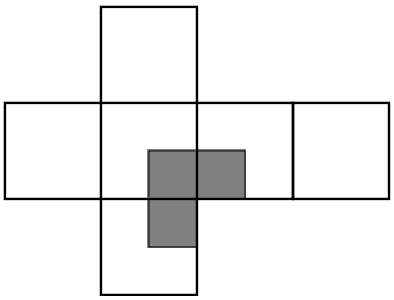


Figure 12. Unfolded Cube Graph for Figure 11.

The masks for each scan consists of the center voxel and each voxel on step backwards along each scanning direction. These masks can be realized in any dimension. In 2D, we need four masks (and four scans), in 3D we need eight masks, in 4D 16 masks etc. The number of neighbours in each mask grows obviously from 2 to 3 to 4 etc. In **n** dimensions, we need 2^n masks, each with center hoxel and **n** neighbours.

For each **n**-dimensional space, a number of different algorithms are possible, all with the high precision that the EDT gives, but still with a few small errors. The more neighbours we use, the fewer errors will we get. The extreme cases are **n** neighbours as mentioned above, which is the fastest, but with the lowest precision (a higher number of hoxels with a small error), and 3^n-1 neighbours, which gives the highest precision. Since the precision of the first, simplest case is good enough for any reasonable application, we no not describe the other cases in detail.

We end this section with a more compact definition of the n-neighbour Corner EDT.

n-dimensional Corner EDT, n neighbours

We have a set of scan directions:
$(d_i, i \in \{1..n\}) \in \{-1, +1\}$

For each set of d_i possible (2^n different ones), we have one scan, where for each center hoxel ($x_1 \ldots x_n$), the following neighbours are used:

$$\sum_{i=1}^{n} (x_1, x_2, \ldots, x_{i-1}, x_i + d_i, , x_{i+1} \ldots, x_n))$$

6. Performance

To evaluate the performance of different algorithms, we should examine key operations like the number of memory accesses and the number of arithmetic operations.

An even simpler measure is the number of masks, which is the number of times the center pixel will be updated. As long as the mask sizes are reasonably small, this is an acceptable simplification.

Another simple measure is the total number of members in all masks. For small masks, this is not so accurate since the processing of the center pixel takes more time than the processing of the neighbours.

Below, a small collection of Euclidean distance transformation algorithms is listed, with the number of scans (number of masks) and the total number of members in all masks used. The calculation of the number of mask members is left out due to space limitations. Algorithms marked with (S) are separable.

Algorithm	Scans	Pixel accesses
2 dimensions:		
8SSED	4	14
3-scan EDT (S)	3	14
3 dimensions:		
6-neighbour EDT	8	22
26-neighbour EDT	8	46
6-neighbour Corner EDT (S)	8	32
26-neighbour Corner EDT (S)	8	64
4-scan (26-neighbour) EDT (S)	4	52
Arbitrary (n) dimensions:		
2n-neighbour EDT	2^n	$3 \cdot 2^n - 2$
3n-neighbour EDT	2^n	$2 \cdot 3^n - 2^n$
2n-neighbour Corner EDT (S)	2^n	$2^n \cdot (n+1)$
3n-neighbour Corner EDT (S)	2^n	4^n

It is evident from the table that the non-separable algorithms have a considerable advantage over Corner EDT on single processor architectures. The algorithms using the smallest number of scans possible (3-scan and 4-scan in the table), however, have much better performance and are good choices for single processor and parallel systems alike.

7. Conclusions

We have presented separable algorithms for computing EDT in 3 or higher dimensions, as well as some principles behind the algorithm design. For this task, we use a new tool, the Unfolded Cube Graph, to visualize the 2-dimensional direction space for 3-dimensional images. This graph proved to be very suitable for the kind of analysis needed.

Among the results are a 4-scan algorithm for 3-dimensional images and an algorithm for arbitrary dimensions.

References

[1] G. Borgefors (1984), "Distance transformations in arbitrary dimensions", *Computer Graphics and Image Processing* 27, 321-345.

[2] G. Borgefors (1986), "Distance transformations in digital images", *Computer Vision, Graphics and Image Processing* 34, 344-371.

[3] P.E. Danielsson (1980), "Euclidean distance mapping", *Computer Graphics and Image Processing* 14, 227-248.

[4] R. Mohr, R. Bajcsy (1983), "Packing volumes by spheres", *IEEE Trans. on Pattern Analysis and Machine Intelligence* 5, 111-116.

[5] U. Montanari (1968), "A method for obtaining skeletons using a quasi-Euclidean distance", *Journal of the ACM* 15, 600-624.

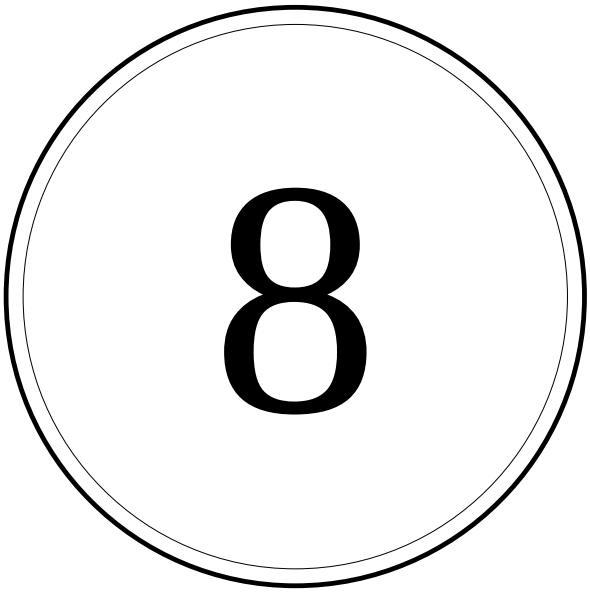
[6] I. Ragnemalm (1990), "Contour processing distance transforms", in: Cantoni et. al., eds, *Progress in Image Analysis and Processing*, World Scientific, Singapore, 204-212.

[7] I. Ragnemalm (1989), "The Euclidean Distance Transform and its implementation on SIMD architectures", *Proceedings, 6th Scandinavian Conf. on Image Analysis*, 379-384.

[8] A. Rosenfeld, J.L. Pfaltz (1966), "Sequential operations in digital picture processing", *Journal of the ACM* 13, 471-494.

[9] H. Yamada (1984), "Complete Euclidean distance transformation by parallel operation", *Proceedings, 7:th International Conference on Pattern Recognition*, 69-71.

[10] Q.Z. Ye (1988), "The Signed Euclidean Distance Transform and its applications", *Proceedings, 9:th International Conference on Pattern Recognition*, 495-499.



Rotation invariant skeletonization by thinning using anchor points

Accepted for 8th Scandinavian Conference on Image Analysis, Tromsø, 1993.

Rotation invariant skeletonization by thinning using anchor points

Ingemar Ragnemalm

Dept. of Electrical Engineering,Linköping University,S-581 83 Linköping, SwedenE-mail: ingemar@isy.liu.se

Abstract:

A thinning algorithm is proposed, using ordered propagation over the Euclidean distance transform using anchor points. Two different methods for extracting anchor points are used. The resulting skeleton is a connected, 1 pixel wide set of pixels along the medial axis. It is rotation invariant within the limits imposed by the sampling grid. The algorithm is also useful for some picture editing operations.

Keywords: Thinning, skeleton, distance transformation, ordered propagation, local maxima, picture editing.

1. Introduction

In the analysis of binary images, a critical step is to transform a binary shape into a compact form more suitable for feature extraction. This is often done by thinning operations, Medial Axis Transformation (MAT), resulting in a skeleton of the shape. The skeleton can be used either as a compressed form of a shape or as a step in the process of analyzing the shape.

The kind of skeleton discussed in this paper is the discrete skeleton, defined as a set of pixels which is a subset of the original discrete shape. This limits the precision to what the discrete grid allows, and makes it hard to make a good error measure for the operation. The alternative is to generate a skeleton of some continuous structures. This can be a subset of the Voronoi diagram, as proposed by Ogniewicz and Ilg (1990), or curve segments, as proposed my Leymarie and Levine (1992).

The term *skeleton* has been used both for the set of *local maxima*, which the set of pixels from which the shape can be reconstructed (see section 4), and for the *connected skeleton*, a one pixel wide structure along the medial axis of the shape. Our goal is to produce the latter structure, but we will use the former as a tool in the process. In the following, the word skeleton will be reserved for the connected skeleton.

Blum (1967) defines the medial axis as all points where the equidistance contour (contour on a fixed distance to the border, wave front) has a corner. In the continuous case, the medial axis is a connected set of infinitely thin lines or curves. This is the ideal skeleton, of which the discrete, pixel-based skeleton is an approximation. See Figure 1 for an example. The task of thinning binary shapes in order to produce such a discrete, connected skeleton is a subject that has been treated in hundreds of papers since the 1960's. For a recent survey, see Lam (1992).

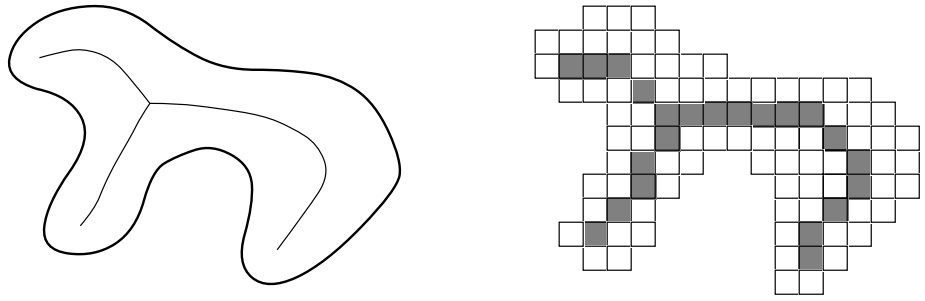


Figure 1. A continuous shape and its medial axis (left) and a discrete approximation with its discrete skeleton (right).

2. Properties of digital skeletons

The desirable properties of skeletons are usually derived from Blum's definition for the continuous case. Blum's definition does not work as it is in the discrete space, since any object has lots of discrete corners rather than the smooth shapes used in Blum's model. Instead, we use a set of properties that should be met by our thinning algorithm, adapted from Hilditch (1969):

1) Thinness; one pixel wide

2) Position; located on the medial axis

3) Connectivity preserving

4) Stability; end point preserving

Verwer (1988) orders these properties in the order 3, 1, 4, 2, where the first is the most important. To the list of desirable properties, we add two that are usually considered slightly less important:

5) Reconstructivity

6) Rotation invariance

Explanations and discussion:

1) The skeleton should be only one pixel wide, to make it easy to track branches.

2) The skeleton should be located in the center of the objects, in order to represent the position of the object in a reliable way. This property is equivalent to the rotation invariance property if the Euclidean metric is used.

3) The topology of both the object and the background should be preserved. An object is not allowed to break up into several skeletons, and the number of holes (genus) should be preserved.

4) Once an end point has been established, it should be preserved. This property is fundamental, since if it is not fulfilled the skeleton will be reduced to a point or set of loops with no branches.

5) Reconstruction of the original shape is usually accomplished by drawing a disk centered in each skeleton pixel, with a radius equal to the maximum distance from the pixel to the border of the original shape (a value provided by a distance transformation). In such a case, the thinness and reconstructivity properties are incompatible, since any pixel in a 2 pixel wide shape can only reconstruct itself. However, there are algorithms that use non-symmetrical disks around the pixels, i.e. in Dougherty and Giardina (1987). In such a case, both 1 pixel width and reconstructivity can be accomplished.

6) For a given shape, the skeleton should have the same structure (with deviations within one pixel distance due to the discrete grid) for all rotated versions of the shape. Perfect rotation invariance is impossible, due to the discrete grid. It is also impossible to meet if the object has edge noise, in which case the problems are due to under sampling. Nevertheless, the property is essential for robust shape analysis.

Our algorithm fulfils all properties except one of either 1) or 5), since they are incompatible as long as we do not consider reconstruction using non-symmetrical disks, as mentioned above. One of these two properties must be left unfulfilled.

3. Skeletons and distance transformations

Most thinning algorithms use simple metrics like City Block or Chessboard distance, e.g. Hildich (1969). The skeletons produced by such algorithms are highly rotation sensitive. A common demonstration if the rotation dependency is to apply these algorithms on a square in different rotations, especially comparing the result at 0 and 45 degrees. See Figure 2.

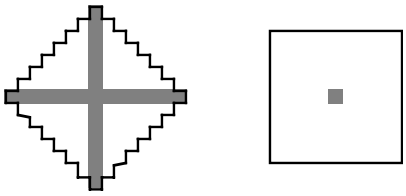


Figure 2. Many skeletonization algorithms are highly sensitive to rotation.

These problems can be reduced by using more accurate metrics. This is usually accomplished by making a *distance map*, produced with a *distance transformation* (DT), of the object to be skeletonized, and then extract the skeleton from the distance map. We can then use accurate metrics like Chamfer metrics (e.g. Verwer (1988)) or the Euclidean metric (e.g. Leymarie and Levine (1992), Klein and Kübler (1987), Ogniewicz and Ilg (1990), Arcelli and Sanniti di Baja (1992)).

Thus, many skeletonization algorithms are based on DTs. Some authors even suggest skeletonization to be the by far most important application of DTs, almost implying equivalence between the two concepts. There are, however, many other applications for DTs.

A distance transformation is an operation that takes a binary image as an input, and generates a greyscale image, the distance map, where each pixel is assigned the distance to the closest background pixel (0 in the case of background pixels). There is a dual case, where the distance denotes the distance to the closest object pixel, and all background pixels get the distance 0, but this case is not useful for skeletonization and therefore not treated further in this paper.

The Euclidean distance transform (EDT), invented by Danielsson (1980), uses vectors instead of scalar distance values. This allows the generation of error-free or practically error-free distance maps by means of fast propagation methods.

The most obvious and apparently robust starting point for generating a rotation invariant skeleton is to extract it from the Euclidean distance transform. However, it is hardly possible to accomplish this by merely applying some simple rule on a small neighbourhood.

In this paper, we will present a new algorithm that produce skeletons using the Euclidean distance transform. This is done with high speed, a time proportional to the size of the image. It starts by finding a set of pixels to use as anchor points. The following sections discuss two different methods to generate this set.

4. The set of local maxima

The set of local maxima was introduced by Rosenfeld (1966) under the name *distance skeleton*. Montanari (1968) proposed an algorithm for the pseudo-Euclidean case. Danielsson (1980) described an algorithm for extracting local maxima from Euclidean distance maps. A related algorithm was suggested by Borgefors et. al. (1991).

It should be stressed that extracting local maxima is not a trivial operation. It can easily be extracted from a distance map using simple metrics like City Block by simply comparing the distance values within a neighbourhood. This is not possible when using Euclidean or pseudo-Euclidean metrics. Both the EDT-based algorithms mentioned above use some kind of look-up table to determine whether a pixel is a local maximum or not.

Using the definitions from Borgefors et. al. (1991), the set of local maxima from the Euclidean distance map is defined as the set of pixels whose associated disk (with the distance value in its center pixel as radius) is not completely covered by any other disk. The disk used is the discrete disk, as exemplified by Figure 3. One discrete disk may cover another even though the corresponding continuous disk does not.

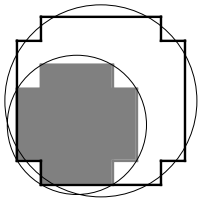


Figure 3. A larger (white) digital disk completely covers a smaller (grey) disk. The corresponding continuous disk (circle) of the larger disk does not necessarily cover the smaller one.

This definition will not produce the truly minimal set of local maxima, since a disk may be covered by two other combined disks, and therefore not necessary for the reconstruction, but still be included in the generated set.

The set of local maxima is usually not connected, so using it directly for skeletonization is not topology preserving. It also often result in structures more than one pixel wide, which adds a pre- or postprocessing step to our algorithm to get a 1 pixel wide skeleton.

5. The α-skeleton

The α-skeleton was suggested by Kruse (1991) as a skeletonization method. In contrast to the local maxima extraction, the method is very elegant and simple to implement, but we find that it is not as robust as the local maxima.

For each object pixel, the Euclidean distance map holds a vector from the pixel to the closest background pixel, i.e. the border. We call such a vector the *EDT vector* of the pixel. This means that along the MAT, we can find neighbour pixels with EDT vectors pointing to different sides of the object.

For each pixel in the shape, calculate the angle formed by its EDT vector and the EDT vector of any neighbour within a given neighbourhood (2·2). The pixel is a member of the α-skeleton if this angle is larger than the threshold angle α. This test can be formulated as follows (adapted from Kruse (1991)):

Let N_i be the EDT vector of each of four pixels located in a 2·2 neighbourhood, as in Figure 4.

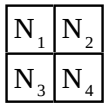


Figure 4. The 2·2 neighbourhood being used.

We find an **I** such that

$N_1 \cdot N_I = \min(N_1 \cdot N_i;\ i = 2,3,4)$

Let N_j be the one of these two vectors with the largest norm:

$N_j := \max(|N_1|, |N_I|)$

In the case when $|N_1| = |N_I|$, we choose $N_j := N_1$. Then, N_j is a member of the α-skeleton if and only if

$$\frac{N_1 \cdot N_I}{|N_1||N_I|} < \cos(\alpha)$$

where α is the chosen threshold.

This method detects a set of pixels along the medial axis of the object. This structure is always one pixel wide. For an intuitive argument for this, see Figure 5. We see that both when the width is an even or odd number of pixels, the skeleton is one pixel wide.

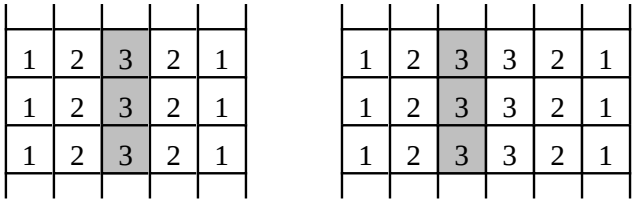


Figure 5. Two shape segments of different width (odd and even width, resp.), both resulting in a 1 pixel wide structure.

However, it is not guaranteed to be connected, and it does not give the possibility to reconstruct the shape. It also has problems with noise in some cases. According to Kruse (1992), this noise can be reduced by using a distance threshold, discarding all pixels under the threshold. With this modification, the α-skeleton has proven useful in certain industrial applications.

An example of this noise problem is given in Figure 6. A pixel that may be assigned any of several vectors in the EDT computation may be classified as a skeleton pixel or not, depending on which vector is chosen. With $\alpha \approx 90^\circ$, the pixel in the figure from which the vectors starts will be classified as a α-skeleton pixel if either of the two lower vectors are assigned to it, but not if any of the two upper vectors are chosen.

These problems are most noticeable close to the border, which is why the thresholding method mentioned above helps. The problem will not be eliminated, but there will be fewer cases where it makes a difference, and the differences will be smaller, less apparent to the human eye.

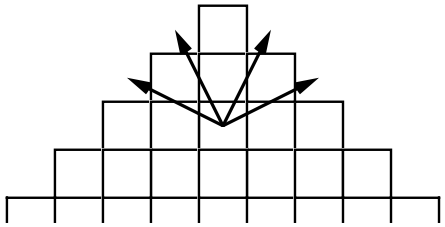


Figure 6. A case where the α-skeleton (in this case with $\alpha \approx 90^\circ$) gives an unpredictable result.

6. Thinning with anchor points

Both local maxima extraction and α-skeleton extraction produce a set of pixels on the medial axis representing the shape. However, neither of them produce connected sets.

It is possible to link a disconnected set by procedures like ridge riding. Here, we will use another approach, namely eroding the object from the edges while preserving anchor points (local maxima or α-skeleton) and connectivity.

The erosion (or rather thinning) is done with an ordered propagation method, where the pixels are accessed in order of increasing distance. Since we have the EDT available, used for generating the set of local maxima or the α-skeleton, we can scan the EDT and sort the pixels according to distance. We use bucket sorting by this, putting pointers to each pixel in a bucket with the appropriate distance value.

Below, we will refer to the pixels at one and the same Euclidean distance as a *layer*. Note that this is usually a small set of pixels, sparsely distributed along the equidistance curve of the object.

For each possible distance value, we inspect each pixel with that distance (in that layer) to see

1) if it is an anchor point or

2) if the topology would change by removing it.

If neither of these conditions is met, the pixel is zeroed.

The test for connectivity, to avoid topology changes, is made using the masks shown in Figure 7 (adapted from Danielsson (1990)). If all neighbours marked 1 are set and all neighbours marked 0 are reset, the center pixel, marked X, can be reset. The figure shows the mask set used for 8-connected objects. Similar sets exist for 4-connected objects.

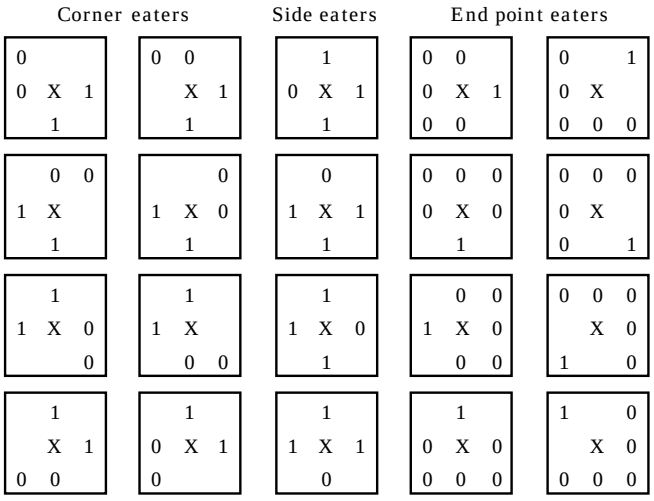


Figure 7. The 20 masks used for determining whether a pixel can be removed without changing the topology of 8-connected objects.

Pixels meeting condition 1) are removed from the bucket immediately, but, of course, not removed from the binary image. For all pixels meeting only condition 2), the test is

repeated until there are no changes. If we only test once, false branches may occur, as exemplified by Figures 8 and 9.

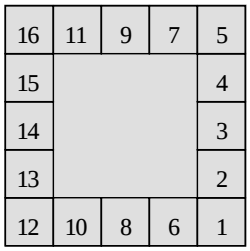


Figure 8. A simple example where the repeated inspection of some pixels are necessary.

The pixels along the border in Figure 8 can typically be accessed in order according to the numbers shown (due to the initial scanning of the image). All the pixels 1 through 14 can safely be removed without changing the topology. However, in the 4-connected case, the pixel 15 can not be removed since pixel 16 would then be disconnected. See Figure 9. Pixel 16 is removed immediately afterwards. This would cause an unwanted branch if we only inspected the border once. Instead, we inspect the remaining pixels again, and repeat this until no more changes occur, in which case we may continue to the next layer (distance value).

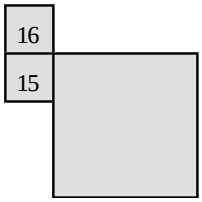


Figure 9. The shape in Figure 8 after pixels 1 to 14 have been removed.

Note that the amount of extra work is minimal, since most pixels are removed in the first pass over a layer, and in most cases there will be no change in the second pass.

Basic thinning algorithm:

```
Scan the EDT, put pointers to each pixel except anchor points in a bucket according to the
squared distance value.
for all buckets
    repeat
        for all pixels in the bucket
            if the pixel is not needed for connectivity
                clear the pixel from the image
                remove the pixel from the current bucket.
    until no pixels were removed during the last loop
    remove all remaining pixels from the bucket
```

Since all pixels must be queued in the bucket structure simultaneously, this method uses more memory than necessary. If the memory consumption must be reduced, the algorithm can be integrated to a more memory efficient one. This algorithm follows below

in pseudo code. For each layer (pixels at the same Euclidean distance to the border), we do three tasks: EDT propagation to the following layers, anchor pixel extraction and connectivity test. When these three tasks are done for one layer, the algorithm process the next layer. The EDT step will generate the EDM for a 3·3 neighbourhood around all the pixels in the layer. Since our anchor pixel extraction methods use either a 3·3 neighbourhood (local maxima) or a 2·2 neighbourhood (α-skeleton) this is enough.

Integrated thinning algorithm:

```
put (pointers to) all object pixels on edges in bucket 1
set the vectors in the EDM for all pixels in bucket 1
b := 1
while nonempty buckets exist do begin
    for all pixels p in bucket b
        propagate EDT to 8-neighbours of p (make pointers in higher buckets)
    for all pixels p in bucket b
        test if p is an anchor pixel
        if p is an anchor pixel, remove it from bucket b
    repeat
        changed := false
        for all pixels in bucket b
            test if b is necessary for connectivity
            if it is not necessary
                remove p from bucket b
                clear the pixel p in the binary image
                changed := true;
    until changed = false
    b := b + 1
end
```

If we use the set of local maxima as anchor points and want a 1 pixel wide skeleton rather than a set from which reconstruction is possible, we must add a pre- or post-processing step where we thin the set of local maxima. This thinning should also be done in order of increasing distance, in order not to displace the skeleton from the medial axis. We did this as a post processing step, removing all the eight end point eaters from the mask set in Figure 7, but adding the four 1-pixel end point eaters in Figure 10.

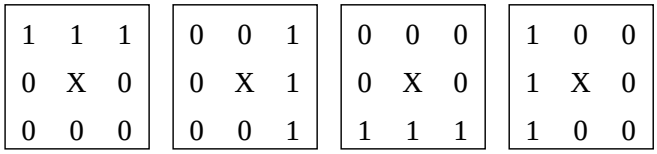


Figure 10. Four 1-pixel end point eaters, replacing the 8 end point eaters (Figure 7) in the post processing thinning step.

This post processing is also necessary for the generation of 8-connected skeletons using the α-skeleton, since the α-skeleton is often a 4-connected structure (when it is connected at all). For generating 4-connected skeletons, however, our experiments indicate that the α-skeleton has no need for this post processing.

Finally, let us make a few comments on efficient implementation of the algorithms described above.

The number of buckets needed for bucket sorting Euclidean distances is very large. Therefore, it can be worthwhile to use the bucket number *modulo* the number of buckets available, as suggested by Ragnemalm (1992). This reduces the number of buckets to a number proportional to the side of the image (n^1) rather than proportional to the total image size (n^2).

The EDT part of the algorithm can be speeded up by using directed masks, as suggested by Verwer (1988) and Ragnemalm (1990). This typically reduces the number of pixels inspected for each center pixel to 2 rather than 8, but adds the overhead of the direction handling.

The check for topology preserving involve a large number of masks for each pixel (Figure 7). Rather than applying these masks by accessing the image 4-6 times per mask, we can form an 8-bit word with the binary value of each neighbour. Then each mask can be tested for with one bitwise AND and one equality test operation.

For example, assume that we have the neighbours stored in a 8-bit word named *sum*, where the least significant bit is the uppermost, and we then continue clockwise. This give the neighbours the values shown in Figure 11. Then, the upper left mask in Figure 7 can be tested for with the following condition:

```
if sum AND (4 + 16 + 64 + 128) = (4 + 16) then... { reset the pixel }
```

128	1	2
64		4
32	16	8

Figure 11. Pixel values used for implementing the masks in Figure 7.

7. Picture editing

Ablameyko et. al. (1989) propose a method for forcing topology changes to a binary image, i.e. splitting or merging objects. When merging, objects closer to each other than a given distance should be joined. When splitting an object, it is done in places where the object is thinner than a given distance. The operations are related to closing and opening operations, respectively, but make smaller changes to the objects. The result should be the same shapes as before the topology change except for a 1 pixel wide area that cause the splitting or joining. See Figure 12.

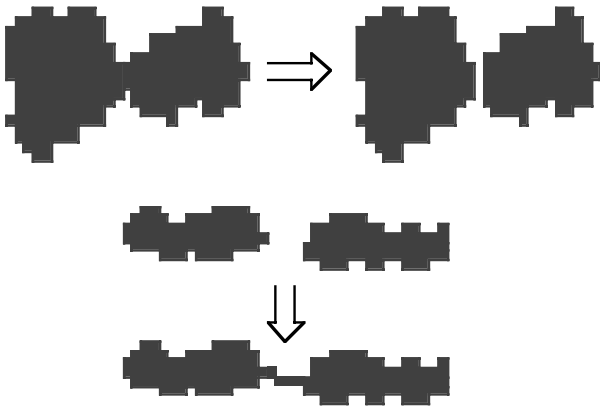


Figure 12. Forcing topology changes to an image without changing the shapes in other ways.

The thinning algorithm described above can be used for this task. For joining objects, we can use the following procedure:

1. Dilate the input image A the desired distance d. An efficient erosion or dilation algorithm suitable for the purpose is described by Ragnemalm (1992). This gives us the dilated image B.

2. Thin the image B using the image A as anchor points. In this case, we should obviously not use the post-processing step where the anchor points are thinned.

The result is an image that is the union of the image A and pixels needed for preserving topology changes caused by the dilation. The algorithm can be modified to splitting by using erosion followed by thinning of the background.

We get the following advantages over the algorithm of Ablameyko et. al. (1989):

• Higher precision. We use the Euclidean metric. However, it is not very hard to modify Ablameyko's algorithm in to use Euclidean metric too.

• Fewer image scans. Using ordered propagation enables us to avoid many image scans. The dilation need one single image scan. The thinning needs one image scan unless we can keep the final Contour List after the dilation. However, we also need to generate the B image as a separate image.

• The shapes of the original objects are preserved. In the Ablameyko's algorithm, the shapes were smoothed in the process, which is a side-effect that may not be desirable.

8. Computer generated examples

In the following figures, the algorithm has been run on a test image (Figure 13) with an assortment of shapes. 8-connective skeletons are generated.

The rotation invariance is demonstrated by the triangular shapes, three of approximately equal size but with different rotation and one larger triangle. In order to demonstrate the correctness with respect to topology, the shapes in the upper left corner of the image were included. Finally, an arbitrary shape with a longer medial axis is included (lower right).

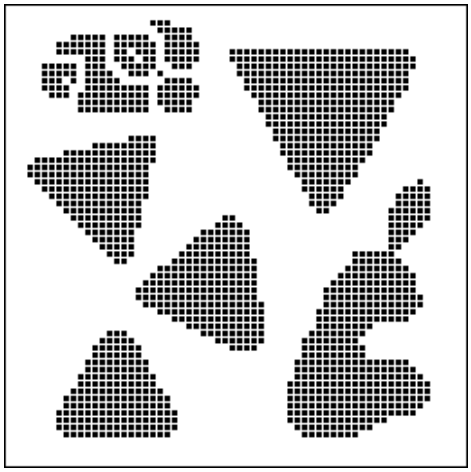


Figure 13. Original binary image with an assortment of shapes.

The left image in Figure 14 shows the local maxima extracted (using EDT) from the test image. Note that the structures are not 1 pixel wide and not connected. The right image in Figure 15 shows the result after applying our algorithm with the local maxima as anchor points, and post processing to achieve 1 pixel width.

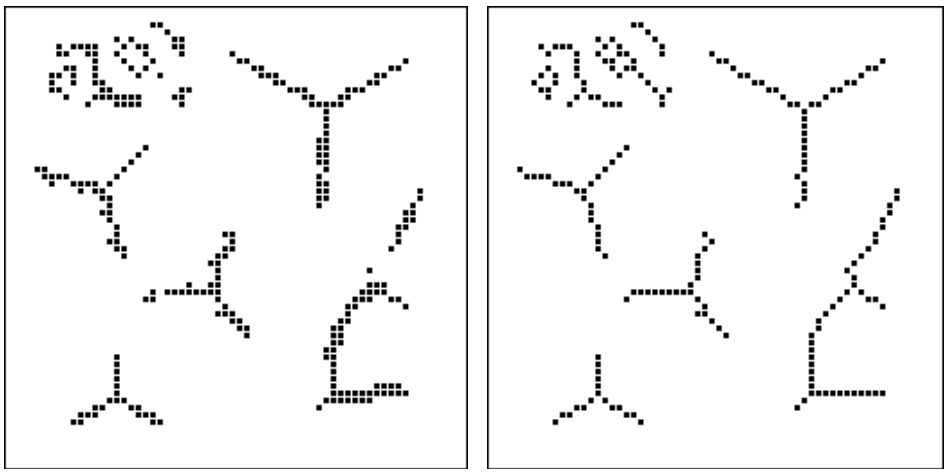


Figure 14. Local maxima extracted from previous image, and the skeletons generated from the local maxima.

Figure 15 shows the corresponding image pair, but for the α-skeleton. The left image shows the α-skeleton of Figure 13. Again, the structures are not connected, but they are always 1 pixel wide. The right image shows the result after applying our algorithm with the α-skeleton as anchor points. This image has been post processed like the previous one, but this time to prune it to 8-connectedness. As mentioned in the previous section, this would not be necessary when generating a 4-connected skeleton.

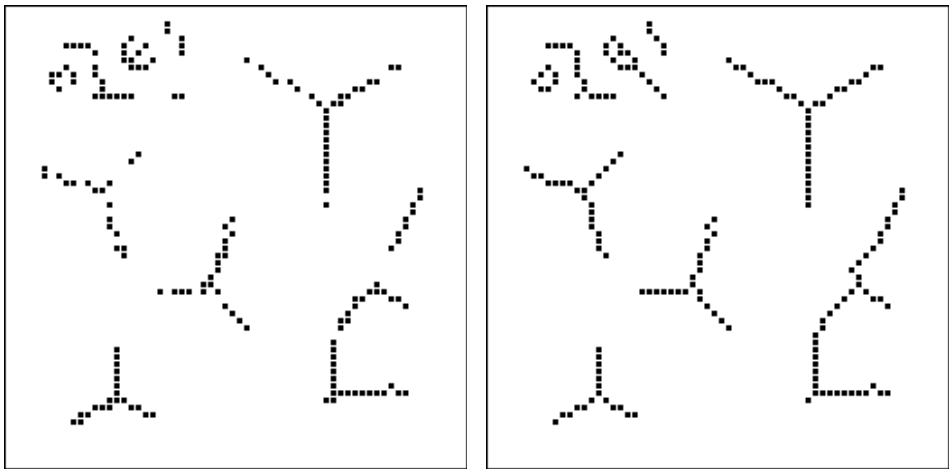


Figure 15. α-skeleton with α=90˚, and the skeletons generated the from α-skeleton.

Skeletons generated from local maxima tend to have some short random branches. These branches occur when single local maxima appear some distance from the medial axis, and are generated from sampling noise at the edges.

Skeletons from α-skeletons, on the other hand, has a different noise problem, as mentioned in section 5. It will cause some branches to arbitrarily appear or disappear when the branch has angles close to the chosen angle α. In our example, this problem does not manifest.

We conclude that both methods give good results. All requirements on the skeletons are fulfilled to reasonable extent.

9. Conclusions

A new thinning algorithm was presented. First, we generate the Euclidean distance map of the binary input image. Second, we extract anchor points from the distance map. Either of two different extraction methods can be used, local maxima or α-skeleton.

The input image was then eroded using an ordered propagation method, preserving connectivity and anchor points. The resulting skeleton is rotation invariant and connectivity preserving. We may get either a 1 pixel wide skeleton or a skeleton from which it is possible to reconstruct the original shape (granted that the distance is kept for each skeleton pixel).

This algorithm can also be integrated so the entire process is done in one propagation process.

References

H. Blum (1967), "A transformation for extracting new descriptors of shape", in: W. Wathen-Dunn, ed., *Models for the Perception of Speech and Visual Form*, Cambridge, MA, MIT Press, pp 362-380.

G. Borgefors, I. Ragnemalm, G. Sanniti di Baja (1991), "The Euclidean Distance Transform: Finding the local maxima and reconstructing the shape", *Proceedings, 7th Scandinavian Conf. on Image Analysis*, pp 974-981.

P.E. Danielsson (1980), "Euclidean Distance Mapping", *Computer Graphics and Image Processing* 14, 227-248.

P.E. Danielsson (1990), *Bildbehandling 1990*, course material for the course "Bildbehandling" at Dept. of EE, Linköping University.

E.R. Dougherty, C.R. Giardina (1987), *Matrix Structured Image Processing*, Prentice-Hall, Englewood Cliffs, New Jersey.

C.J. Hilditch (1969), "Linear Skeletons from Square Cupboards", *Machine Intelligence 4*, Univ. Press, Edinburgh, pp 403-420.

B. Kruse (1991), "An exact sequential Euclidean distance algorithm with application to skeletonizing", Proceedings, 7th *Scandinavian Conf. on Image Analysis*, pp 982-992 (revised version of an Internal Report, Teragon Systems, 1987).

B. Kruse (1992), personal communication.

F. Leymarie, M.D. Levine (1992), "Simulating the Grassfire Transform Using an Active Contour Model", *IEEE Trans. on Pattern Analysis and Machine Intelligence 14*, pp 56-75.

F. Klein, O. Kübler (1987), "Euclidean distance transformations and model-guided image interpretation", *Pattern Recognition Letters 5*, pp 19-29.

U. Montanari (1968), "A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance", *Journal of the ACM* 15, pp 600-624.

R. Ogniewicz, M. Ilg (1990), "Skeletons with Euclidean metric and correct topology and their application in Object Recognition and Document Analysis", *Proc. 4th Int. Symposium on Spatial Data Handling*, Zürich, pp 15-24.

I. Ragnemalm (1990), “Contour processing distance transforms”, in: Cantoni et. al. eds. *Progress in Image Analysis and Processing, World Scientific, Singapore,* pp 204-212. (A related paper is published in CVGIP/IU.)

I. Ragnemalm (1992), “Fast erosion and dilation by contour processing and tresholding of Euclidean distance maps”, *Pattern Recognition Letters* 13, pp 161-166.

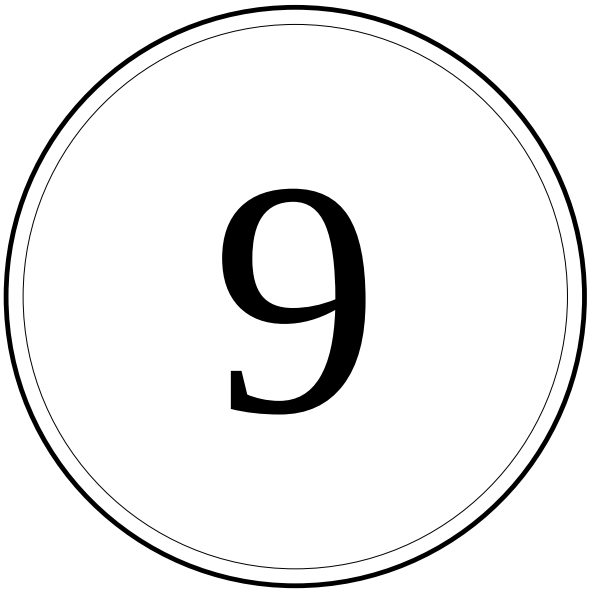
A. Rosenfeld, J.L. Pfaltz (1966), “Sequential Operations in Digital Picture Processing”, *Journal of the ACM* 13, pp 471-494.

B.J.H. Verwer (1988), “Improved Metrics in Image Processing applied to the Hilditch Skeleton”, *Proc. 9:th International Conf. on Pattern Recognition*, Rome, pp 137-142.

S. Ablameyko, C. Arcelli, G. Sanniti di Baja (1989), “Using Distance Information for Editing Binary Pictures”, *Proc. 6th Scandinavian Conf. on Image Analysis*, Oulo, pp 401-407.

L. Lam, S.-W. Lee, C.Y. Suen (1992), “Thinning Algorithms - A Comprehensive Survey”, *IEEE Trans. on Pattern Analysis and Machine Intelligence 14*, pp 869-885.

C. Arcelli, G. Sanniti di Baja (1992), “Ridge points in Euclidean distance maps”, *Pattern Recognition Letters 13*, pp 237-243.



Towards a minimal shape representation using maximal disks

Not yet published.

Towards a minimal shape representation using maximal disks

Ingemar Ragnemalm
Dept. of Electrical Engineering,
University of Linköping,
S-581 83 Linköping,
SWEDEN

Gunilla Borgefors
Swedish Defence Research Estab.
Box 1165,
S-581 11 Linköping,
SWEDEN

Abstract:

From a distance map of a digital shape, a set of maximal disks can be extracted from which it is possible to recover the shape completely. The set should ideally be as small as possible, but the commonly used algorithms do not produce the minimal set. Our experiments show that the number of disks in the minimal set can be substantially reduced for most shapes.

1. Introduction

In a distance map, produced by a distance transformation, each pixel tells the distance to the closest background pixel. This implies that the biggest disk that can be drawn completely within the object, centered on a given pixel, is the disk with a radius equal to the distance value in that pixel.

It is possible to find a *subset* of these disks, from which the complete shape can be recovered. This operation is used in binary image processing in order to generate a compact representation of the shapes in the image, which can be used for shape analysis or compression. It was first introduced by Rosenfeld [8] and developed further by Montanari [5].

The subset was originally called *distance skeleton*, since it is located on the medial axis of the shapes. It is, however, not topology-preserving, so it is not a skeleton in the usual sense. Later, it has been called the *set of local maxima*, which refers to how the set can be extracted from distance maps with simple metrics. With more accurate metrics (Euclidean or pseudo-Euclidean metrics), though, that name is misleading. See further below. In this paper, we choose to call it the *set of Necessary Maximal Disks* (NMD), and the pixels to be identified, previously called local maxima, are the *centres of necessary maximal disks* (CNMD).

The NMD gives us a complete description of shapes in the image, with the possibility to reconstruct the shapes. Hence, it can be used for data compression. Since the set gives information about what the most significant parts of the shapes are, it is useful for shape analysis. We can also use the set as a first step when generating connected skeletons [7].

In this paper, we state that the common methods for extracting the NMD result in sets that are far bigger than the minimal ones, and propose a method for finding sets that are minimal or close to minimal. We will concentrate on the Euclidean metric in this paper, but the results are relevant for any discrete metric.

2. Digital disks

The NMD is found by first applying a distance transform on the binary image, producing a distance map where each feature pixel is assigned the distance to the closest non-feature pixel. Many authors have discussed the distance transform concept. [1-9]

We let each pixel in the distance map define a *disk*. We distinguish between the *continuous disk* (cdisk) and the *digital disk* (ddisk). The cdisk is a continuous shape centred on a pixel centre. For each cdisk, there exists a ddisk, which is the set of pixels inside the cdisk, that is, if the cdisk has radius **d**, the ddisk is the set of pixels with centres on a distance < **d** from the central pixel. See Figure 1.

The metric, i.e. distance transform, defines the shape of the cdisk. For City Block distance, the cdisks are diamond-shaped. For Chessboard distance, cdisks are squares. For weighted (Chamfer) distances, the cdisks are polygons of higher order. Finally, for Euclidean distance, the cdisks are circular. Figure 1 shows examples for these four cases.

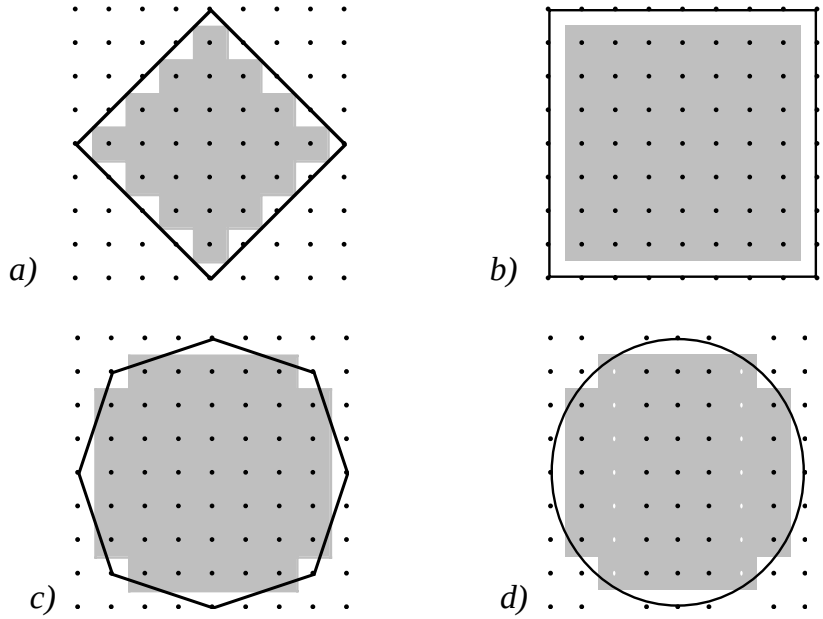


Figure 1. The continuous disks and digital disks defined by a pixel with distance value 4 in the distance map (radius 4). a) City Block distance. b) Chessboard distance. c) Chamfer 3-4 distance. d) Euclidean distance (see also Figure 2).

A pixel and its distance value define the location and size, respectively, of a cdisk, which in turn defines a ddisk. Using distance values from a distance map, all such ddisks are completely within the object and touches the border in at least one point.

The smallest ddisks from a Euclidean distance map are shown in Figure 2. When designing algorithms for extraction of maximal disks from distance maps with high precision (Euclidean or pseudo-Euclidean), we must be aware of the irregular shapes these disks have. For pseudo-Euclidean metrics, the NMDs are defined in [1,9] (although not by that name).

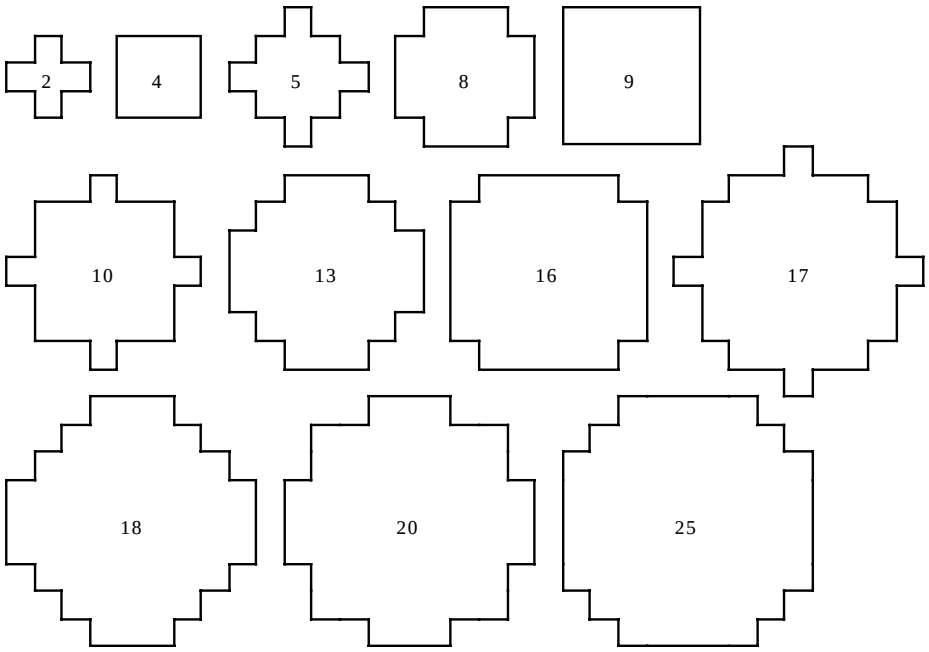


Figure 2. Digital disks for Euclidean distance. The numbers are the squared radii for each disk.

3. The standard set of necessary maximal disks

The NMD is a set of disks, from which it is possible to reconstruct the original shape. We want this set to be as small as possible. Thus, we reformulate the sentence above and define the following:

Definition 1. The *Minimal set of Necessary Maximal Disks* (MNMD) for a distance map of a shape is the minimal set of disks, from which it is possible to reconstruct the original shape.

For practical reasons the standard methods define the set of maximal disks in a different, somewhat weaker, way:

Definition 2. A maximal disk, defined by the location and distance value of its centre pixel, belongs to the *Standard set of Necessary Maximal Disks* (SNMD) if its ddisk is not completely covered by any one other ddisk centered on one of the neighbours of the centre of the disk.

This is the definition used by Borgefors et. al. [2], but reformulated to the terminology used in this paper. With this definition, it is possible to extract the NMD from the distance map with fast algorithms.

In simple metrics (i.e. City Block or Chessboard distance), these pixels can, as the name *local maxima* implies, easily be found as the local maxima in the distance map, and thus be extracted by a simple operator. This simple method does not work for weighted metrics or the Euclidean metric. See [1,2,9]. Also, it is not suitable to extract SNMD by checking whether the cdisk of one pixel is covered by the cdisk of a neighbour. The reason for this is that though the ddisk of the centre pixel completely covers the cdisk of the diagonal neighbour, this is not the case for the cdisks, as illustrated in Figure 3.

In the Figure, we see two cdisks and their ddisks. The larger ddisk covers the smaller one completely, but the larger cdisk does not cover the smaller one. Thus, the smaller disk would be extracted as a necessary disk if we only checked whether the cdisk was covered. The resulting NMD would be unnecessarily large.

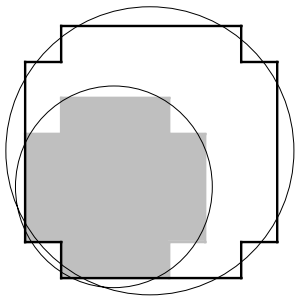


Figure 3. One ddisk may cover another ddisk even if the corresponding cdisk does not cover the smaller cdisk.

Instead, lookup tables are used. Danielsson [4] uses a 2-dimensional lookup table, which is also used in the reconstruction process. The extraction method proposed by Borgefors et.al. [2] uses a 1-dimensional lookup table. Similar problems appear for many weighted metrics, making the task of extracting the SNMD as hard as in the EDT [9].

When we want to reconstruct the original shape from the maximal disks, the task is essentially to paint the digital disk for each maximal disk, but since large areas - actually most areas - are covered by more than one disk, we should do it in a more efficient way. We can use propagating methods, similar to those used for distance transformation [2], called a *reverse* distance transformation. Note that the time required for reconstruction with such methods is constant (proportional only to the image size), independent on the number of maximal disks.

4. Optimizing the set of maximal disks

The standard set of necessary maximal disks (SNMD) generated by the methods described above have one serious drawback. The definition of SNMD (Definition 2 above) removes any pixel with a ddisk covered by one other ddisk if that ddisk is defined by a feature pixel among the 8-neighbours of the pixel in question, but not when a ddisk is a subset of the union of several other ddisks, even though the ddisk is not necessary for reconstruction in such a case.

The reason for this drawback is, of course, that it is not possible to, in an efficient way, detect when a ddisk is covered by several others, and we have no method to find the absolutely minimal set. The question we may ask ourself is: Is it worthwhile to find a smaller set? Is the optimal set much smaller than the set we can extract with the known, efficient methods, or is the difference marginal?

In order to answer these questions, we will now describe a method to find a set that is much closer to the true minimum. It will usually find the minimum, but we can not guarantee it. The method is not fast compared to the common extraction methods. We call the set extracted by this method the *Optimized set of Necessary Maximal Disks* (ONMD), not to be confused with the *minimal* set (though they are usually identical).

Let us look at a simple example, Figure 4. Since we concentrate on Euclidean distance maps in this paper, we use the squared Euclidean distance, even though the shape of the disks do not differ from Chessboard when applied to the small shape in Figure 4.

1	1	1	1	1	1	1	1
1	4	4	4	4	4	4	1
1	4	4	4	4	4	4	1
1	1	1	1	1	1	1	1

Figure 4. A small rectangle, Euclidean distance map (squared distance) and the CMDs of the SNMD (shaded).

In the Figure, 12 pixels form the SNMD (shaded in Figure 4). Eight of these, all but the four end pixels, have ddisks that are completely covered by the ddisks of its two horizontal neighbours. Therefore, we can weed out most of them. The maximal disks remaining after optimization (the ONMD) are shown in Figure 5.

1	1	1	1	1	1	1	1
1	4	4	4	4	4	4	1
1	4	4	4	4	4	4	1
1	1	1	1	1	1	1	1

Figure 5. Centres of disks in the optimized set of maximal disks for Figure 4.

To find this smaller set, we use the following algorithm:

Algorithm to find the optimized set of necessary maximal disks:
1. Initially, extract the standard set of necessary maximal disks (SNMD as defined above). We call this set I.
2. Initiate a new image, B, to zero. For each pixel in I, add 1 to the value of all pixels in B covered by the corresponding ddisk.
3. For each pixel in I, in order of increasing distance value, inspect all pixels in B covered by the ddisk. If all pixels hold values ≥ 2, remove the pixel from I and decrement all pixels in B covered by the ddisk.

This is the basic algorithm for finding the ONMD. Note that for Euclidean metric, when we have to scan over all pixels in a ddisk, we can not do this with the usual circle drawing algorithms used in Computer Graphics. Those common circle drawing algorithms generally take the enclosing rectangle as input. We have to be able to distinguish between all different ddisks possible, as shown in Figure 2, e.g. disks 10, 13 and 16, which an enclosing rectangle is insufficient to do. For other metrics, the disks have other shapes, as previously illustrated in i Figure 1, which will demand other kinds of routines.

In step 3, we should access the CNMDs in order of increasing distance value. It should be noted that sorting of distance values, even Euclidean ones, in a digital image can be done with bucket sorting [6], and is therefore very fast.

To return to the example in Figure 4, the image B generated in step 2 is shown in Figure 6, where each pixel has a count of the number of ddisks covering it.

1	2	3	3	3	3	2	1
2	4	6	6	6	6	4	2
2	4	6	6	6	6	4	2
1	2	3	3	3	3	2	1

Figure 6. Image B, the number of ddisks covering each pixel.

In step 3, all the maximal disks in the I set are tested in an arbitrary order (since all have the same distance). The rightmost and leftmost CNMD have ddisks that include one of the corners, where there is a 1, so they can not be removed. Any other maximal disk has 2 or more in all pixels. Figure 7 shows two of the disks (shaded). One CNMD (left) can not be removed, but the other one (right) can, since there are no 1’s in the ddisk.

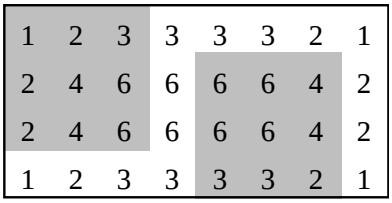


Figure 7. Two of the ddisks tested in step 3 (shaded). The leftmost include a 1, so it must not be removed. The rightmost one include only pixels with a count of 2 or more, so we can remove it and decrement all pixels inside the ddisk.

According to Figure 7, the four CNMDs closest to the corners in Figure 5 are necessary. Each one of the two in the middle, however, have been chosen arbitrarily among two, depending on what order they are visited.

Figure 8 shows a more complicated example. It is an elongated shape with one rounded end, one rectangular end and a noise pixel on one side. The set of shaded pixels, both light and dark ones, is the SMD, the initial set I. Framed pixels are CNMDs that include pixels with count = 1 in the B image, i.e. CNMDs that must be included in the final NMD. Out of the remaining maximal disks in SNMD, most can be removed, but during the optimization, more 1's will appear, indicated that some more must be in the final ONMD (dark shaded pixels without frames). These pixels can usually be chosen differently.

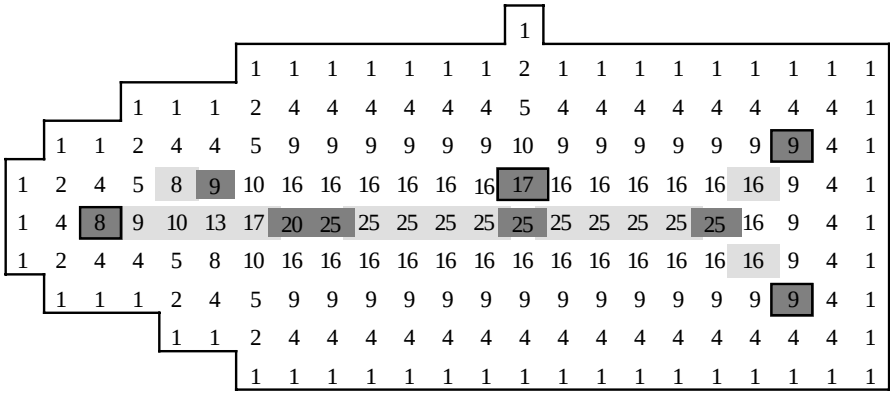


Figure 8. A shape with its CNMDs. SNMD are shaded pixels, ONMD dark shaded ones.

The proof that these ddisks cover this original shape is left as a cut and paste exercise to the reader (Figure 9). For this particular example, the number of maximal disks was reduced from an SNMD with 24 pixels to an ONMD with 9, which is a significant reduction. As we will see in the following computer-generated examples, this big reduction is not an isolated occurrence, but something that is actually possible in most cases.

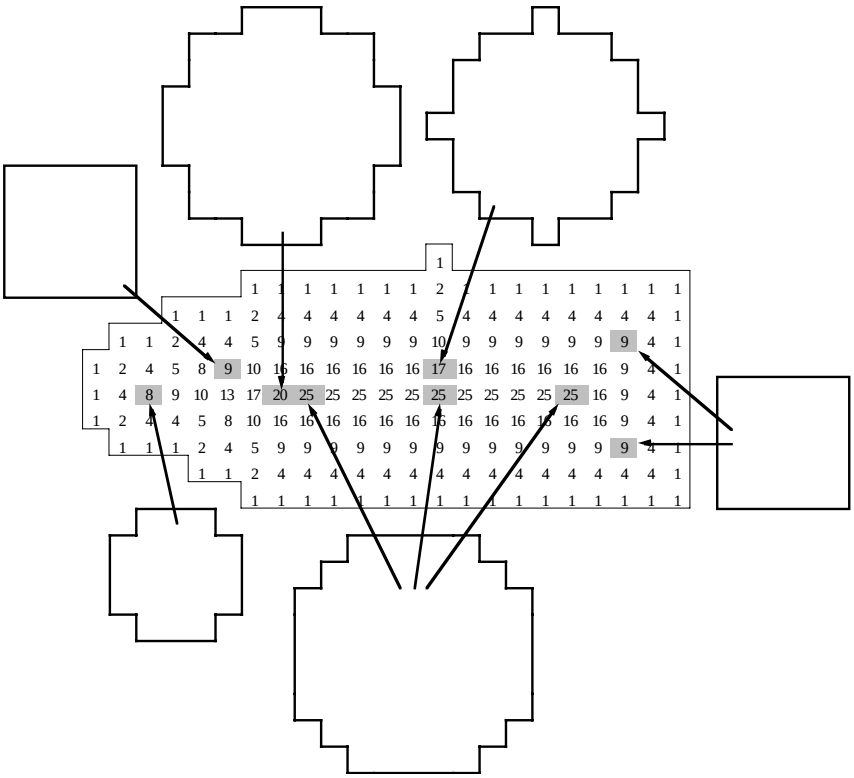


Figure 9. The previous figure but with the appropriate ddisks shown.

5. Expected reduction rates

So, what is the reduction in the general case? The SNMD and MNMD sets are obviously strongly data dependent in size and placement. This data dependency makes it impossible to make any exact statements, but we can study some special cases.

We first consider some cases where optimization has no effect, namely when SNMD = MNMD. The first special case is *objects shaped like the ddisks being applied*. These objects will have an SNMD with only a single disk per object, with is already minimal. A second special case is *very thin objects*. If an object has width ≤ 2 pixels, both the MNMD, SNMD and ONMD are the entire shape, since no bigger disks can fit in it. However, experiments indicate that very few other shapes have an SNMD that is identical to the MNMD.

As a third, more interesting special case, consider a *straight, elongated object* with width d and length L, where L>>d. The number of maximal disks before optimization is proportional to L, except for the ends where we get an end point effect depending on the width and the shape on the ends. With L>>d, as assumed above, we can neglect this end point effect.

In this case, a horizontal or vertical object with an odd width will get a SNMD with L pixels (approx.), while an object with even width will get 2L pixels. For a diagonally oriented object, we get $L/\sqrt{2}$ and $\sqrt{2}L$, respectively. We claim that the ONMDs of objects like this are substantially smaller than the SNMDs, and that the reduction will be bigger for wider objects.

Consider a horizontal object with odd width. It has one maximal disk per pixel distance of length. See Figure 10.

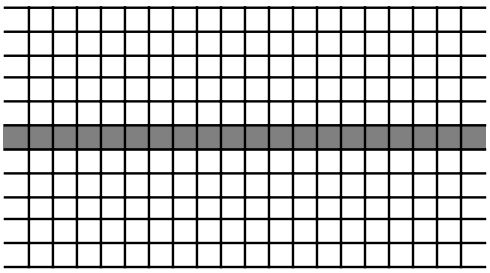


Figure 10. A horizontal, straight, long object has one maximal disk per pixel distance of length. (Pixels in the SNMD are shaded.)

The reduction from the SNMD to ONMD will depend on how big part of the edge that one disk can cover. In this case, the reduction can be calculated. Figure 11 shows the cdisk for one CNMD in the shape. It reaches one pixel distance outside the shape, and has radius d/2 + 1 = r+1 for a shape with width d.

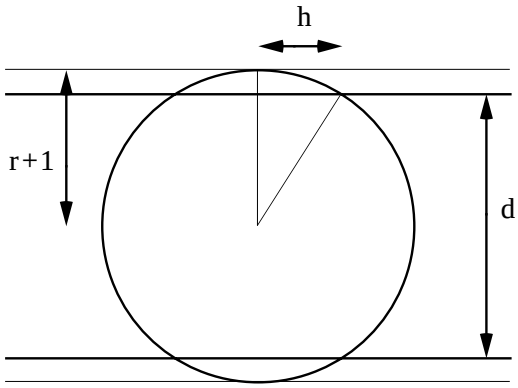


Figure 11. How to calculate the width of the edge of a disk in the horizontal case.

The disk will cover the width 2h of the edge of the shape. More exact, the number of pixels covered along the edge is $2\lceil h \rceil - 1$ (where $\lceil\ \rceil$ rounds the argument up to the next integer). From Figure 11, it follows that

$$(r+1)^2 = r^2 + h^2 \quad \Rightarrow \quad h = \pm\sqrt{2r+1} = \pm\sqrt{d+1}$$

Thus, the number of pixels covered by each disk is $2\lceil \sqrt{d+1} \rceil - 1$. This is the reduction ratio achieved by optimization. We can expect a reduction of the same order for other directions as well, though it is very difficult to calculate it exactly for most directions. The calculation above suggests that it will be proportional to $\sqrt{d}$.

For simple metrics like Chessboard and City Block distance, we get a different behaviour. We will get very high reduction ratios for shapes that are aligned with the disk edges, but low or no reduction in other cases. Figure 12 shows examples for Chessboard distance.

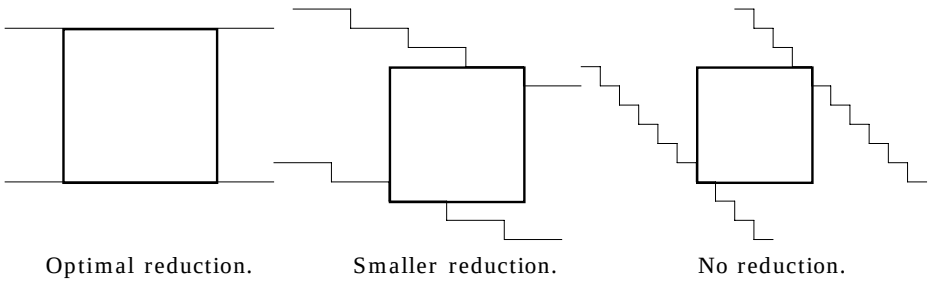


Figure 12. For simple metrics, the direction of object edges affect the possible reduction of maximal disk.

In the example, the reduction ratio from SNMD to ONMD varies from the object width d to 1 (no reduction). For City Block distance, we will get similar results, but with the biggest ratio when Chessboard has the lowest and vice versa. When varying the direction of the object from horizontal or vertical to diagonal, City Block will gradually give more reduction while Chessboard will give less.

This suggests that we can expect the Euclidean and pseudo-Euclidean disks to give a less rotation dependent result (just like they do in many other problems), which should result in a representation that is more significant for the shape. Comparing the effects of different metrics on this problem is, however, not pursued further here but is left for future work.

6. Computer generated examples

In the following figures, Figures 13-16, the algorithm has been run on a test image with four triangles with different rotation, and then another test image with four squares. We have used the Euclidean metric for these experiments. In both cases, the improvement from optimization is considerable, despite the fact that the objects are rather compact and thereby close to disk-shaped.

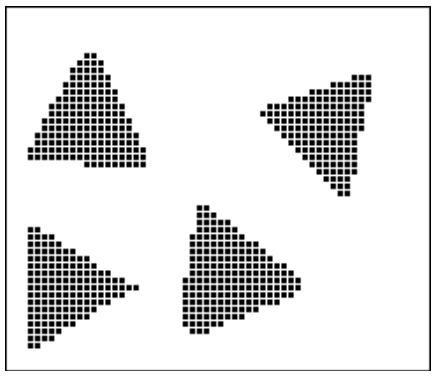


Figure 13. Original image with four triangles.

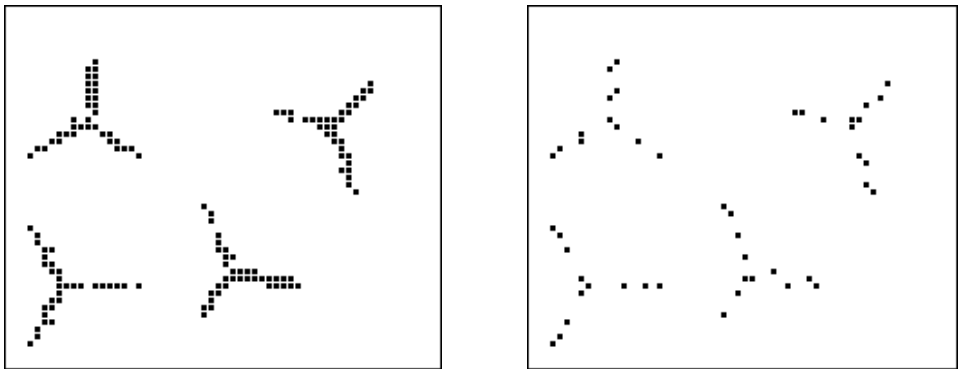


Figure 14. Standard set of maximal disks (SNMD) extracted from Figure 13, and the optimized set (ONMD) (49 out of 140 remain).

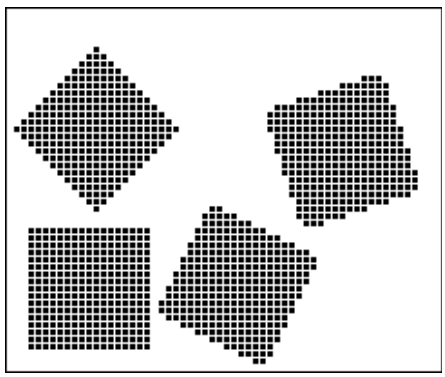


Figure 15. Original image with four squares.

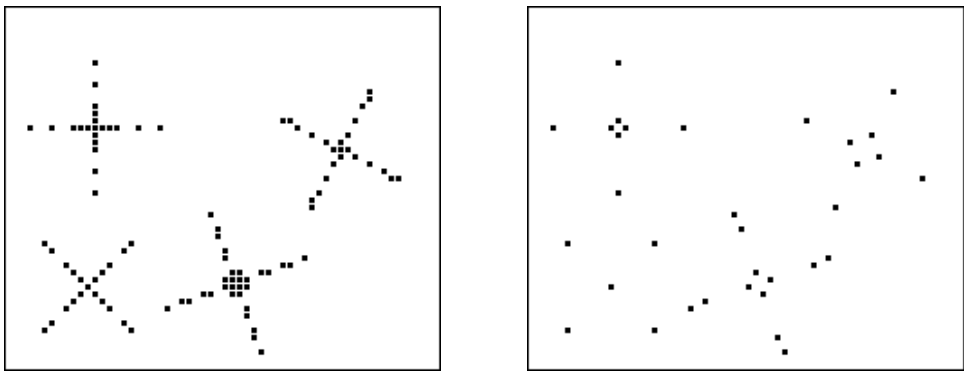


Figure 16. Standard set of maximal disks (SNMD) extracted from Figure 15, and the optimized set (ONMD) (33 out of 99 remain).

From our experiments, we conclude that the standard set of maximal disks (SNMD) is usually far from optimal, and though our algorithm is not very efficient, it shows that we have reason to search for other, more efficient algorithms performing the task of finding the opimized sets.

7. Conclusions

We found that the standard set of necessary maximal disks (a.k.a. set of local maxima) for a shape is very far from the optimal one. An algorithm was proposed that is able to find a set of necessary maximal disks that is optimal or close to optimal. In our experiments, the generated set was typically between half and a third of the size of the set of necessary maximal disks generated by the previously known, faster algorithms. We have shown that we can expect higher ratios for bigger objects.

We conclude that it is possible to find much smaller sets than the previously known algorithms could find. We have presented an algorithm that finds a set that is much closer to the minimum. The algorithm can be used for any metric, though our examples only use the Euclidean metric. The algorithm is slow, but if the objective is compression, this may be worthwhile. The reconstruction is as fast or faster than using other methods.

References

[1] C. Arcelli, G. Sanniti di Baja, "Weighted Distance Transforms: A characterization", *Image Analysis and Processing II,* Plenum Press 1988, pp 205-211.

[2] G. Borgefors, I. Ragnemalm, G. Sanniti di Baja, "The Euclidean Distance Transform: Finding the local maxima and reconstructing the shape", Proceedings, *7th Scandinavian Conf. on Image Analysis*, 1991, pp 974-981.

[3] G. Borgefors, "Distance Transformations in Digital Images", *Computer Vision, Graphics and Image Processing 34*, 1986, pp 344-371.

[4] P.E. Danielsson, "Euclidean Distance Mapping", *Computer Graphics and Image Processing 14, 1980*, pp. 227-248.

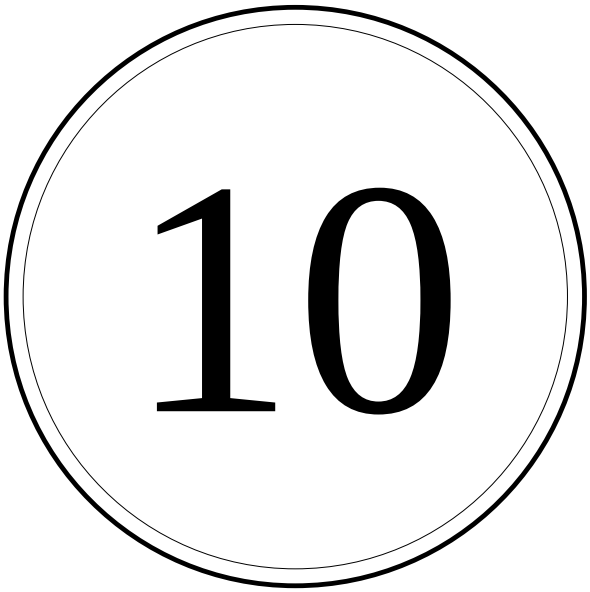
[5] U. Montanari, "A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance", *Journal of the ACM, vol 15 No 4*, October 1968, pp 600-624.

[6] I. Ragnemalm, "Fast erosion and dilation by contour processing and tresholding of Euclidean distance maps", *Pattern Recognition Letters* 13, 1992, 161-166.

[7] I. Ragnemalm, "Rotation invariant skeletonization by thinning using anchor points", accepted to *8th Scandinavian Conf. on Image Analysis*, 1993.

[8] A. Rosenfeld, J.L. Pfaltz, "Sequential Operations in Digital Picture Processing", *Journal of the ACM*, Vol 13, No 4, October 1966, pp 471-494.

[9] G. Borgefors, "Centres of Maximal Discs in the 5-7-11 Distance Transform", accepted to *8th Scandinavian Conf. on Image Analysis*, 1993.



A note on “Optimization on Euclidean Distance Transformation Using Grayscale Morphology”

Submitted to Journal of Visual Communication and Image Representation.

A note on “Optimization on Euclidean Distance Transformation Using Grayscale Morphology”

Ingemar Ragnemalm

Dept. of EE, Linköping University, S-581 83 Linköping, Sweden

Abstract

Shih et. al. propose a method for optimizing the inner loop of the Euclidean Distance Transform, using simple and fast arithmetics for the calculation of distance values during distance propagation. Two parallel and one sequential algorithm using this method were proposed. In this note, we show why the sequential algorithm fails, while the two parallel algorithms produce correct results.

1. Introduction

A *distance transformation* is an operation that takes a binary image as input and generates a grayscale image, a *distance map*, where each background (object) pixel holds the distance to the closest object (background) pixel. Using only one scalar value per pixel, fast distance mapping algorithms using neighborhood operations will always have errors proportional to the distance [6, 7, 8].

This problem was solved with the Euclidean Distance Transform (EDT), first proposed by Danielsson [1]. The difference from the original distance transforms is that it uses vectors rather than scalar values. This makes it possible to produce Euclidean distance maps in short time, but also to create a vector map, additional information that is useful in many applications [5].

The sequential 8SSED algorithm [1] produces error-free distance maps except for a few single points where small errors occur. The errors are very small, 0.09 pixel distances or less, which is usually negligible. The parallel algorithms are error-free, as shown by Yamada [4]. Considering that the errors in the sequential EDT are so small and do not grow proportionally to the distance, it is not an approximation to Euclidean distance in the same sense as the pseudo-Euclidean algorithms [6,7]. Error-free sequential versions of EDT have also been developed [10].

Shih and Wu [3] propose a different method to produce Euclidean distance transforms, using no vectors but rather the squared distance plus a propagation number. They propose two parallel algorithms and one sequential algorithm, the Double Two-scan Algorithm (DTA for short). On close inspection, it turns out that DTA produces distance maps with significant errors. The errors are due to several problems not addressed in [3], which are discussed below. However, it turns out that none of these problems affect the parallel algorithms, so they will actually produce the same output as the original PED algorithm [1].

Before discussing the algorithms, we will recall the well-known concept of *Voronoi diagrams*. In a Euclidean distance map, each pixel will hold the distance to the closest object pixel. This means that for any object pixel, we can find a polygon that encloses all points for which the chosen object pixel is the closest one. This polygon encloses a set of pixels, possibly only the object pixel itself. In the EDT, this set of pixels should all get distance values referring to the chosen object pixel. The polygon is the *Voronoi polygon* for the object pixel. The set of all Voronoi polygons is the Voronoi diagram of the image. See e.g. Guibas et. al. [9].

2. The sequential algorithm

Shih and Wu [3] derive their algorithm using grayscale morphology. We will, however, describe their methods in a more traditional way, using simple geometrical properties. The distance transform is done using square distance values ($D^2(p)$ below), a common approach to avoid real numbers during EDT computation. The distance value is updated using the number of propagation steps (L(p) below).

Definitions:

$D^2(p)$ is the squared distance value stored in the pixel p.

L(p) is the propagation number calculated for the pixel p (see below).

$D^2_{cand(n)}(p)$ is a candidate value for $D^2(p)$, given by the neighbor n.

When processing a pixel p, we use a set of pixels M - the *mask* - in a 3x3 neighborhood around p. In the parallel algorithm the entire neighborhood is applied at once, while in the sequential DTA algorithm, the neighborhood is split in two parts, applied at different scans. In the following sections, only DTA will be discussed.

For each neighbor pixel in M, we calculate a candidate value. Let h be a 4-neighbor to p, that is a horizontal or vertical neighbor, while d is a diagonal neighbor (8-neighbor but not 4-neighbor). Then, the candidate values calculated for h and d are:

$$\text{horizontal or vertical neighbor to p:} D^2_{cand(h)}(p) = D^2(h) + 2 \cdot L(p) - 1 \quad (1)$$

$$\text{if m is a diagonal neighbor to p:} D^2_{cand(d)}(p) = D^2(d) + 4 \cdot L(p) - 2 \quad (2)$$

Then, p is assigned the minimal distance value for all neighbors i:

$$D^2(p) := \min(D^2_{cand(i)}(p)) \quad (3)$$

These updating rules are summarized in Figure 1. The figure shows the two recursive scans, both in the grayscale morphology structuring element representation used in [3] (matrixes in brackets in Figure 1) and a slightly modified form, replacing l with L(p) to stress that the propagation count is pixel dependent (lower part of the figure). The matrixes (masks) shown in the figure are the M mask used in each scan, as mentioned above. It shows the values to be added to the squared distance value of each neighbor of the center pixel p to produce a candidate value for p.

L(p) is calculated for each of the two scans as a separate distance map with Chessboard distance in the directions used in the masks in Figure 1. (See the next section.)

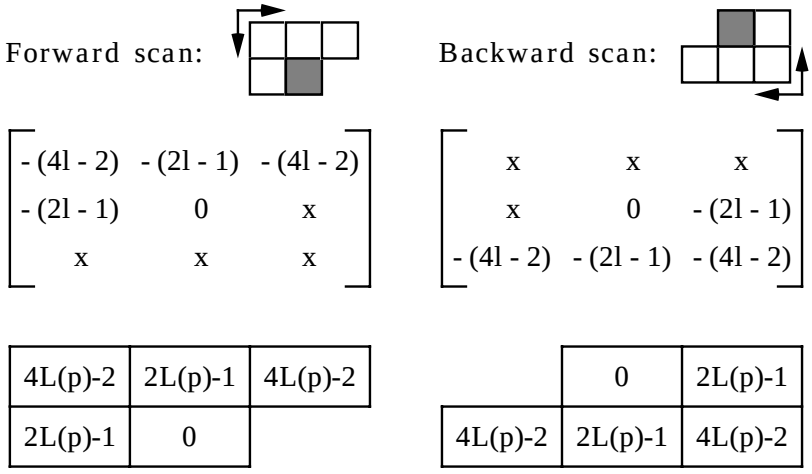


Fig. 1. The complete 3x3 mask with distance offset.

Calculating candidate values for each pixel in a mask, and writing the smallest to the center pixel, is the common approach, used in the original EDT [1]. The difference from most other EDT algorithms is the calculation of the distance values.

3. Euclidean distance versus Chessboard distance

The first, and perhaps most evident problem with DTA is the calculation of the L(p) values. It is done for each scan with a mask with the same shape as the mask being used for the EDT, but with weight 1 in all positions. This results in a distance map with Chessboard distances in these directions. This approach does not work since:

• The closest pixel in Euclidean distance is often not the same as in Chessboard distance.

• For certain kinds of images, the Chessboard distance mapping used will not reach all pixels. (E.g. an image with only a single object pixel.)

For an example of the first error, see Figure 2. The pixel in the lower left corner is 4 pixel distances from the closest pixel in Euclidean distance, and should therefore be assigned the squared distance 16. However, the closest pixel using Chessboard distance is only 3 pixels away. This causes the updating to be done with L(p) = 3 rather than 4, which results in a too small value.

In Figure 2, two updating paths are marked with a) and b). For each of these, the following calculations are made to get the candidate value:

a) $D^2_{cand(h)}(p) = D^2(h) + 2 \cdot L(p) - 1 = 9 + 2 \cdot 3 - 1 = 14$

b) $D^2_{cand(d)}(p) = D^2(d) + 4 \cdot L(p) - 2 = 8 + 4 \cdot 3 - 2 = 18$

The b) value correctly refers to the originating pixel, while a) is wrong, since L(p) should have been 4.

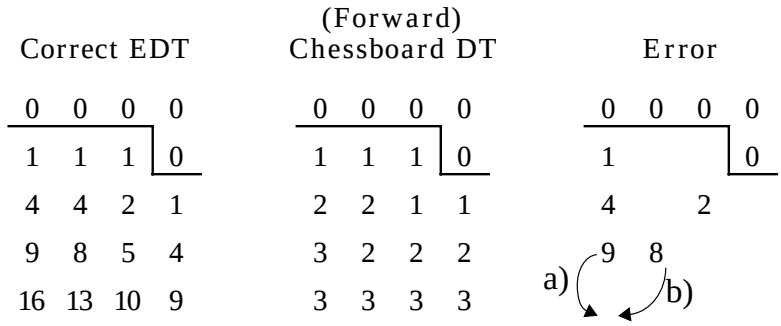


Fig. 2. An example where the L(p) calculation cause errors.

This problem is, however, easily corrected by generating L(p) simultaneously with the EDM, propagating L(p) values when pixels are updated, and using the values in the neighbors (that is, L(h)+1 and L(d)+1) rather than the center pixel for calculating candidate values. From now on, we will assume that this problem is corrected in all examples.

4. Propagation limitations imposed by the distance value updating method

For finding the next problem, we will need to understand why and when the distance calculation in (1) and (2) works. In the following, we will derive these formulas using fundamental geometric properties – essentially just the length of a vector.

Let $v_m = (m_x, m_y)$ be the vector from a pixel m to the originating object pixel (distance zero). The Euclidean distance is then $D(m) = (m_x^2 + m_y^2)^{1/2}$ and the squared distance is $D^2(m) = m_x^2 + m_y^2$. Note that while these vectors are present in most other EDT algorithms, we only have $D^2(m)$ available here.

We also know the number of propagation steps L(m), which is the chessboard distance to the (in Euclidean distance) closest object pixel, that is $L(m) = \max(|m_x|,|m_y|)$. In the parallel algorithm, the propagation number L(p) is the iteration number, while in DTA, it is stored separately.

Let us take the situation in Figure 3. The pixel p is to be updated by calculating candidate values from the neighbors d and h. Any other neighbors may be used as well, but granted that the originating pixel o is the object pixel that is closest to p, only d and h can give rise to the minimal candidate value from the 8-neighborhood of p.

With $v_p = (p_x, p_y)$ being the vector from p to o, we consider only the case where $p_x > 0$, $p_y > 0$ and $p_x \geq p_y$. This is no loss of generality, since the results can be mirrored to all other directions.

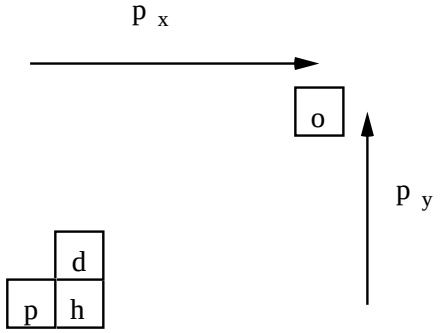


Fig. 3. The p is the pixel to be updated, and d and h are two neighbors one step towards the originating object (zero distance) pixel.

For the neighbors h and d, we have the vectors $v_h = (h_x, h_y)$ and $v_d = (d_x, d_y)$. Then, the candidate value for p calculated from h is:

$$D^2_{cand(h)}(p) = (h_x+1)^2 \cdot h_y^2 = h_x^2 + h_y^2 + 2h_x + 1 = \quad = h_x^2 + h_y^2 + 2(h_x+1) - 1 = D^2(h) + 2(h_x+1) - 1 \qquad (4)$$

Since $p_x \geq p_y$, $p_x = L(p)$. Since p and h are horizontal neighbors, $p_x = h_x + 1$, and we get

$$D^2_{cand(h)}(p) = D^2(h) + 2(h_x+1) - 1 = D^2(h) + 2L(p) - 1$$

which is equivalent to (1). Since $p_x = h_x + 1$, we also see that $D^2_{cand(h)} = D^2(p)$, so this is the correct value on $D^2(p)$, granted that o is the closest object pixel.

For the diagonal neigbor d, we first consider the special case where $p_x = p_y$:

$$D^2_{cand(d)}(p) = (d_x+1)^2 + (d_y+1)^2 = 2\cdot(d_x+1)^2 = 2d_x^2 + 4d_x + 2 == 2d_x^2 + 4(d_x+1) - 2 = D^2(p) + 4(d_x+1) - 2 \quad (5)$$

which, with $p_x = d_x + 1$ and $p_y = d_y + 1$ is equivalent to (2), and $D^2_{cand(d)} = D^2(p)$, so it is a correct value for $D^2(p)$.

When $p_x > p_y > 0$, however, we get:

$$D^2_{cand(d)}(p_{xy}) = (d_x+1)^2 + (d_y+1)^2 = d_x^2 + d_y^2 + 2d_x + 2d_y + 2 \quad (6)$$

which is not the same as (2). Hence, (2) will result in too large candidate distance values anywhere it is used except on diagonal lines from the originating pixels. By subtracting (6) from (5), we find that the difference is $2d_x - 2d_y$. However, since the generated value is too large, as long as there is a neighbor in the 4-neighborhood that will deliver the correct value to the center pixel, the result will be correct. This is not always the case.

The incorrect values generated from diagonal neighbors when not on a diagonal from the originating pixel imply that an algorithm using this scheme must use the propagation paths illustrated in Figure 4.

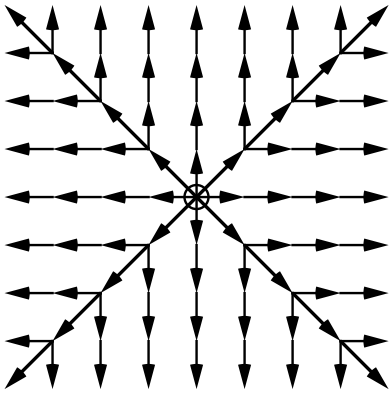
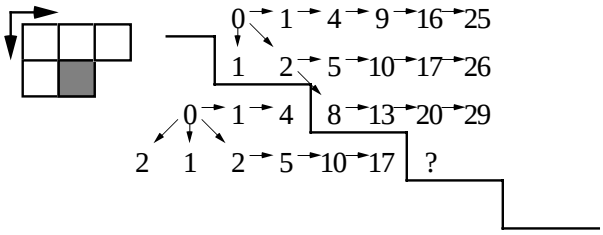


Fig. 4. Only the propagation paths in the Figure generate correct values.

In some cases, this will work, but there are many configurations where this will fail. Figure 5 shows a simple case. In the Figure, the propagation from an object pixel must pass through an area closer to another object pixel (that is, through the Voronoi polygon of another object pixel). If the propagation from the other pixel reaches that area first, the propagation will be cut off.

The lower part of the figure shows what candidate values are generated. The only value that is the correct distance value to any object pixel is the 26 referring to the lower object pixel. For the neighbor above the center pixel, it is questionable what L(p) value to use, but in neither case (CB: Chessboard distance, or P: number of propagation steps) the value will be low enough to change the result in this example.



13+4·4-2 = 27	P:20+2·5-1=29 CB:20+2·4-1=27	29+…
17+2·5-1 = 26	?	

Fig 5. Due to the limited propagation paths, the pixel marked “?” will get the wrong value. The only path that could generate the correct value passes through pixels that are closer to the lower object pixel.

The region that can not be reached will instead get distance values that refer to some other, more distant object pixel, or get updated according to (2) in cases where it isn’t valid, and thereby get a too large value.

5. Propagation limitations imposed by the neighborhoods used

In the previous section, we saw how the algorithm failed to produce a correct result since it produced incorrect distance values in certain directions. Moreover, the masks proposed, illustrated in Figure 1, are poorly chosen and will by themselves cause even more errors.

As pointed out in [2], in a Euclidean distance transform, the pixels that should be reached by the propagation from some object pixel p are the pixels within the Voronoi polygon around p. Moreover, a distance propagation process is guaranteed to work while in the proper Voronoi polygon, even if another propagation reaches the area first. Since Voronoi polygons can sometimes be very elongated in any direction, this guarantee will only hold if for any direction there exist a scan where the direction is supported to infinite distance. If this is not fulfilled, the propagation from each object pixel must reach the paths it needs to use before the propagation from any other object pixel. (See section 5.)

Danielsson [1] used a four-scan algorithm to support propagation in all directions, and in [2], a three-scan algorithm was proposed. Note that Danielsson's algorithm uses masks that are combined two by two into two *double* scans. Figure 6 shows the masks used in these algorithms. It is impossible to support all directions with only two masks.

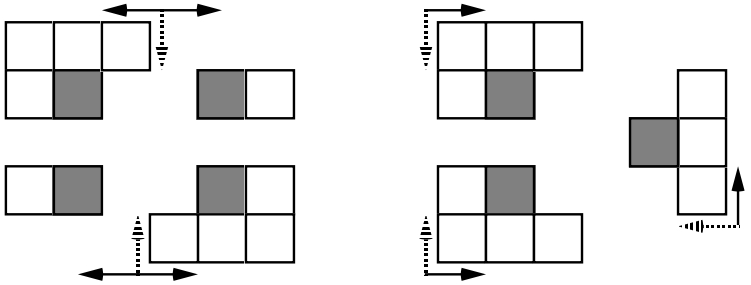


Fig. 6. The masks used for sequential EDT in [1] resp. [2].

The DTA, however, does use only two scans, and does not support propagation to infinity in all directions. This will not only result in the errors mentioned in [2], but will fail even in trivial cases, as illustrated in Figure 7, a case with only a single object pixel. The summations above the figure show how each of the updates along the arrows are generated.

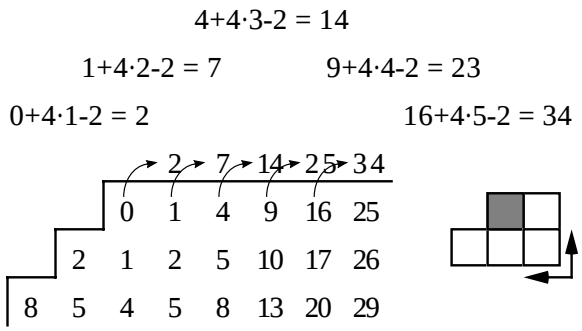


Fig. 7. Bold numbers are incorrectly generated in the second scan.

This problem can easily be solved by using any of the neighborhood sets in [1] or [2]. Disregarding the other problems discussed above, such an algorithm would only have the negligible errors where Voronoi polygons have very acute angles, as pointed out by Danielsson. [1]

6. The parallel algorithms

As we have seen, the sequential algorithm, DTA, does not work properly and should not be used. However, the parallel algorithms proposed does not suffer from these problems. In this section, we will point out under what circumstances they do work, and why.

Two different parallel algorithms were proposed. The first, the 1D algorithm (1DA for short), first scans each row to produce 1-dimensional distance maps, a trivial though sometimes very time consuming operation. Second, for all pixels that have received a distance value (i.e., all pixels for which there is any object pixel on the same row), the distance is squared. Then, 1x3 masks are applied in parallel over the image, using the update scheme described in section 2, using only vertical neighbors and using the iteration number as propagation count L(p).

It is of critical importance that this operation is done in parallel, since the method, like the incorrect sequential algorithm, relies on propagation being able to pass through pixels for which the originating pixel is the closest one. When implemented in parallel, however, this is possible.

Figures 8 and 9 illustrate the propagation process. Figure 8 shows the initial, horizontal distance transformation from three object pixels. Note that when two object pixels are located on the same row, the Voronoi polygon will have a vertical edge between them, if the polygons touch.

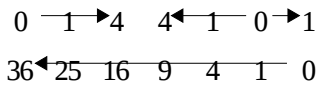


Fig. 8. The initial, horizontal propagation from three object pixels.

The propagation from the pixel on the lower row in Figure 8 will obviously have to pass through the Voronoi polygon of the middle pixel in order to reach pixels far down to the left. In the vertical propagation, this propagation will always be one step ahead of the propagation from the middle pixel, since the pixels are updated in parallel, as illustrated in Figure 9.

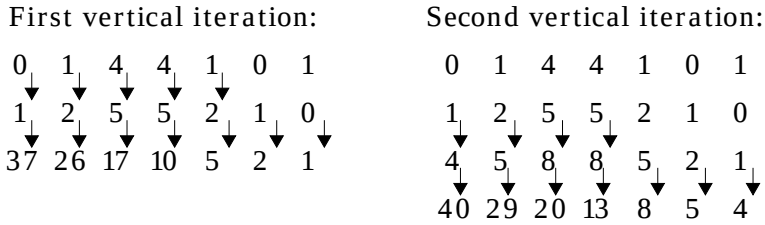


Fig. 9. If the vertical propagation is done in parallel, several propagation processes will queue up after each other without overwriting.

Hence, the 1DA will not suffer from the propagation limitations discussed in section 4, and since the iteration number is used for the propagation step number L(p), the error described in section 3 will not appear.

Since 1DA requires parallel operation to work correctly, it will be far slower than any sequential algorithm (like 8SSED [1]) when implemented in parallel. However, even on a parallel hardware, it is not a good choice, since the initial, horizontal distance transform will have to be implemented in hardware, and it will in the worst case (a single object pixel at the end of a row) require a number of iterations equal to the width of the image. For a parallel hardware, older parallel EDT algorithms [1, 4] or the algorithm described below are better choices.

The second parallel algorithm, the 2D Iterative Erosions Algorithm (2DIEA for short) will work for essentially the same reason. The 2DIEA simply applies the update rules (1) and (2) in a complete 3x3 neighborhood.

The valid propagation paths will still be restricted to the ones shown in Figure 4. However, this will not cause any errors. See Figure 5. The upper object pixel must be at least one pixel distance to the right of the lower pixel, which means that the propagation from that pixel will be at least one pixel distance ahead of the other. Thus, it will be able to propagate through the Voronoi polygon of the other pixel before that area receives its final values.

7. Discussion

According to the arguments in sections 3 to 5, we can expect significant errors when using the sequential DTA algorithm. While some of these problems can be solved, the problems discussed in section 4 do not appear to have a simple solution. DTA neither gives us a well-defined metric nor errors with an upper bound. Hence, it should not be used.

The parallel algorithms, however, do produce Euclidean distance maps, and are quite useful on massively parallel architectures. While the 1DA has worst cases where it is unnecessarily slow, the 2DIEA only require the same number of iterations as previously proposed parallel EDT algorithms [1, 4], but with simpler arithmetics, resulting in higher speed.

A drawback with 2DIEA in some cases is that it can not produce vectors as output, which the algorithms by Danielsson [1] and Yamada [4] do. This makes it less useful for certain applications, while it remains a good choice for applications where only the distance value is of interest.

However, it is quite possible to modify the previously known EDT algorithms to get performance that is close to what the algorithms in [3] promise. The essence of these algorithms is then to update the distance values incrementally, which is a known trick to avoid multiplications. Ye [5] proposes a related method, but using the vector components rather than a special propagation step number. This follows immediately from (4) and (6) when the vector components are known.

An alternative, also proposed by Ye [5], is to use lookup tables to convert vector components to distance values. Such a lookup table should take each vector component as index to a 2-dimensional array, returning either the distance or the squared distance. This will not only remove the need for multiplications, but will also remove the square root computation, and remove the extra storage needed in both the incremental methods mentioned above. This is perhaps the most attractive method, keeping the vector images and producing the distance with no calculations except additions. If the size of the lookup table is prohibitive, we should use incremental methods like the 2DIEA or as proposed above.

8. Conclusions

We have pointed out some problems with the sequential, raster scanning algorithm proposed in [3]. We have also noted that the problems do not affect the proposed parallel algorithms, and demonstrated why. Still, one of the parallel algorithms proposed is of interest when using massively parallel architectures.

However, even the parallel algorithm is of questionable value, since it does not give vectors as output, which the original EDT does [1, 4, 5], but rather just a distance value per pixel. It is usually better to use the lookup table driven method proposed by Ye [5] to achieve higher speed.

References

[1] P.E. Danielsson, "Euclidean Distance Mapping", *Computer Graphics and Image Processing* **14**, 1980, pp. 227-248.

[2] I. Ragnemalm, "The Euclidean Distance Transform and its Implementation on SIMD Architectures", *Proceedings, 6th Scandinavian Conf. on Image Analysis*, Oulo, 1989, pp. 379-384.

[3] F. Shih, H. Wu, "Optimization on Euclidean Distance Transformation Using Grayscale Morphology", *Journal of Visual Communication and Image Representation* **3**, 1992, pp. 104-114.

[4] H. Yamada, "Complete Euclidean Distance Transformation By Parallel Operation", *Proceedings, 7:th International Conference on Pattern Recognition*, 1984, pp. 69-71.

[5] Q.Z. Ye, "The Signed Euclidean Distance Transform and Its Applications", *Proceedings, 9:th International Conference on Pattern Recognition*, 1988, pp 495-499.

[6] G. Borgefors, "Distance Transformations in Digital Images", *Computer Vision, Graphics and Image Processing* **34**, 1986, pp. 344-371.

[7] U. Montanari, "A Method for Obtaining Skeletons Using a Quasi-Euclidean Distance", *Journal of the ACM* **15**, 1968, pp. 600-624.

[8] A. Rosenfeld, J.L. Pfaltz, "Sequential Operations in Digital Picture Processing", *Journal of the ACM* **13**, 1966, pp. 471-494.

[9] L. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams", *ACM Transactions on Graphics* **4**, 1985, pp. 74-123.

[10] I. Ragnemalm, "Neighborhoods for Distance Transformations Using Ordered Propagation", *CVGIP: Image Understanding* **56**, 1992, pp. 399-409.

PostScript error (--nostringval--, findresource)