UTRECHT UNIVERSITY

DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES

Multidimensional Distance Transforms using the FEED-class Algorithm

BSC
COMPUTER SCIENCE

Author:
Sam van der Wal (3962652)

Supervisor:
dr. dr. Egon L. van den Broek

August 28, 2015

Abstract

The Fast Exact Euclidean Distance transform (FEED) algorithm is extended beyond two dimensions. 3D-FEED is introduced, followed by nD-FEED. The nD-FEED algorithm uses the inverse approach from the naive DT algorithms and works with exact euclidean distances. The time-complexity of nD-FEED is proven to be exponential in the amount of dimensions: $\Omega(dN \times 2^d)$.
Keywords:n-dimensional distance transforms, Euclidean distance, FEED, 3D-FEED.

1 Introduction

A distance transform(DT) calculates for a set of pixels in a picture the minimal distance to another set of pixels in a picture. Often, background and object pixel sets are distinguished. Those distance transforms calculate a new image with the distances for each background pixel to the closest object pixel. The resulting image is called a distance map. DT's are used in many scientific or industrial area's like bio-medical image analysis and robotics. The general equation for the distance transform on pixel p to an object pixel $q \in O$ is defined as

$$D(p) = \min_{q \in O} d(p, q), \tag{1}$$

where we use $d(p, q)$ to denote the distance between pixel p and q.

Distances

Calculating distance transforms can be done with different distances metrics, important distances are:

- *Manhattan distance:* The Manhattan distance is the sum of the distance traveled horizontal and vertical direction in a grid. Formula (2) calculates the Manhattan distance which is also known as the *city block distance*.

$$d(p, q) = |p_1 - q_1| + |p_2 - q_2|, \tag{2}$$

- *Chessboard distance:* The chessboard distance is the distance it takes a king on a chessboard to reach a certain point. With formula (3) you can calculate the chessboard distance.

$$d(p, q) = max(|p_1 - q_1|, |p_2 - q_2|), \tag{3}$$

- *Euclidean distance:* The Euclidean distance is the direct distance between two points and is also known as the *Pythagorean distance*. Calculating this distance is however computationally harder than the Manhattan distance or the chessboard distance. Formula (4) calculates the Euclidean distance. In this paper we focus on the Euclidean distance.

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}. \tag{4}$$

Derived from these distances we find how the fundamental distance notation for the Euclidean and Manhattan distance can be defined by the L_u distance metric:

$$d_u(p,q) = (\sum_{i=1}^{n} |p_i - q_i|^u)^{\frac{1}{u}}, \quad (5)$$

where p and q are n-tuple's and i is used to denote their n coordinates (or dimensions). This distance is also known as the *Hinkowski* distance. The Manhattan and Euclidean distance metrics are respectively denoted by L_1 and L_2

Euclidean distance transforms

The Euclidean DT can either be calculated as a *Baseline DT*, where the Euclidean distance is very crudely calculated, an *Approximate Euclidean DT*, where the Euclidean distance is approximated to speedup distance calculations, or an *Exact Euclidean DT*, where the exact Euclidean distances is calculated. Because the exact Euclidean distance uses a square-root evaluation, it is computational harder to calculate the exact DT. We define the exact Euclidean distance as:

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2}, \quad (6)$$

where p and q are points in the n-dimensional space and p_i and q_i are their positions in dimension i.

Pixel-processing

Based on the order in which pixels are processed, there are 4 types in which DT algorithms can be classified[15][7]:

- *Raster scanning*(RS): For each dimension, each pixel is processed from start to end and in reversed order. n-dimensional masks can be used to speed up the process.
- *Ordered propagation*(OP): This method is similar to fire spreading through grass. For each object pixel we start by propagating the distance to each "fire-edge" pixel until we visit the edge of the picture or until we visit another fire front.
- *Independent scanning*(IS): For each dimension, each pixel is processed from start to end. Further propagation of distances of previously encountered object pixels are discarded since the new encountered object pixels are always closer. This process is also done in reversed order.
- *FEED class*(FEED): The FEED algorithm takes a new fundamental approach. Instead of p selecting the minimum distance to $q \in O$, the FEED algorithm feeds for each $q \in O$ the distance to p, which implies that the FEED algorithm is the inverse function of (1). The feed on pixel p is only accepted when the distance to q is lower than the currently known distance to p.

Dimensions

The FEED algorithm works on 2 dimensions, however there are also images which use 3 dimensions (e.g. medical images of the brain, air traffic models). If additional information (e.g. time or speed) needs to be added in the information of the images, more dimensions are needed to illustrate the image. Currently 4 algorithms are available which can calculate exact distances in images with n-dimensions. Table (1) gives a small overview of the history of DT algorithms with their exactness, the way pixels are processed and on how many dimensions they apply. [15]

Algorithm	Year	Exact	PP	Dim
RosenFeld and Pfaltz [11][12]	1966	Baseline	RS	2
Borgefors [1][2]	1984	Baseline	RS	n
Coiras, Santamaria, and Miravet [5]	1998	Baseline	OP	2
Danielsson [6]	1980	Approximate	RS	2
Danielsson (Modified) [3]	1980	Approximate	RS	3
Ye [18]	1988	Approximate	RS	2
Shih and Wu [16]	2004	Approximate	RS	2
Maurer, Jr, Qi and Raghaven [10]	2003	Exact	IS	n
Coeurjolly and Montanvert [4]	2007	Exact	IS	n
Lucet [9]	2009	Exact	IS	n
Felzenszwalb and Huttenlocher [8]	2012	Exact	IS	n
Schouten and van den Broek [15]	2004	Exact	FEED	2

Table 1: An overview of some algorithms for different DT's [17] with respectively the algorithm, the year, the exactness, the way pixels are processed and on how many dimensions. [15].

This paper will introduce the FEED [14][13][17]algorithm as described by Schouten and van den Broek [15], in section 3 the FEED algorithm will be adapted for 3 dimensions, and in section 4 for n-dimensions. The principles of the FEED algorithm will be explained in section 2. The new FEED algorithms will have its theoretical time complexity explained in section 5. In section 6, additional research suggestions are made and this paper closes with a discussion in section 7.

2 Principle of FEED

Let us first define a binary image I, a set of object pixels $O\{o|o \in I \wedge o = 1\}$ and a set of background pixel $B\{b|b \in I \wedge b = 0\}$. Instead of $p \in B$ selecting the minimum distance to $q \in O$ (1), the FEED algorithm feeds for each $q \in O$ the distance to all $p \in B$. This is done with the following algorithm:

Algorithm: Basic FEED

```
foreach q ∈ O do
    foreach p ∈ B do
        D(p) = min(D(p), d(q, p))
    end
end
```

Since this is a very basic approach, several speedups are needed for faster execution time.

2.1 Speedups

Border pixels

The number of feeding object pixels can be significantly reduced if the object pixels that are completely surrounded by other object pixels do not feed. All neighboring object pixels are always closer to background pixel in that direction. An object pixel is a border pixel if one of its four connected neighbors is a background pixel, meaning formula (7) is satisfied, which means that

$$p \in B(O) \iff \exists y | y \in n(p) \wedge y \notin O \wedge p \in O, \tag{7}$$

where $n(p)$ is the set neighboring pixels having a Manhattan distance to p of one. $B(O)$ is the set of border pixels.

Bisection Line

The number of background pixels an object pixel feeds, can also be significantly reduced. Assume feeding pixel p and pixel $q \in O$ and $p \neq q$. For convenience reasons we create a new coordinate system with p in the origin. In order to lower the amount of pixels that are being fed by p, a bisection line can be created between pixel p and q, defined by:

$$2q_1x + 2q_2y = (q_1^2 + q_2^2) \tag{8}$$

In order to select q, a line-scan from p in a specified direction is performed. The scan-line is defined as $(i, j) = k(m_i, m_j)$, with m_i and m_j being integers and k being the running index starting at one. This line-scan stops when it hits another object pixel or the border of the image. Performing a line-scan ensures that the selected object pixel q creates an optimal bisection line in direction (m_i, m_j). High values of m_i or m_j tend to create bisection lines farther away from p then low values.

Perpendicular to this scan-line the bisection line is created. The bisection line splits the image in two parts. The side closer to p will be fed by p and the side closer to q will be fed by q Multiple scan-lines from p can be initiated creating a convex hull of background pixels which will be fed by p.

Bounding Box

Since it is hard to keep track of the convex hull around p when using multiple bisection lines, a bounding box can be created around the convex hull. This bounding box is possibly reduced in size by each new bisection line. In figure (1) a new bisection line l is created by the use of scan-line s towards object pixel q inside bounding box *bb*. Due to the direction of s, only the bounds $b1$ and $b2$ of the bounding box can be moved towards s, which excludes $b3$ and $b4$ from being updated. This results in a new bounding box which excludes the gray parts. If the bounds need to be adapted, then the x-bound needs to be updated to

$$bb.x_{max} = min(bb.x_{max}, max(y_{max}.x, y_{min}.x)). \tag{9}$$

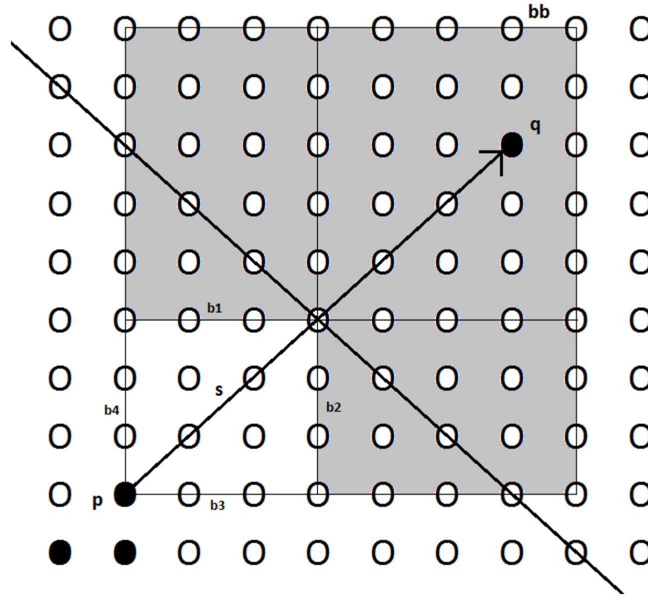

Figure 1: Beginning with a bounding box bb, a line-scan s, originating from border pixel p, is conducted, in the direction of object pixel q. When s hits q, a perpendicular bisection line l is created.

This equation is similar for the y-coordinate. Although the bounding box is often larger than the convex hull, it is computational more efficient to keep track of only the bounding box and not the convex hull. For each row, the bisection lines are used to determine which background pixels should be fed by p. Because of this, only the background pixels that are inside the convex hull are actually fed by p. In the ideal case, two intersecting bisection lines can also update a bound of the bounding box.

3 3D-FEED

When we increase the amount of dimensions to three, the basic FEED [13]algorithm keeps the same inefficient approach. Formula 4 has to be adapted to 3D space:

$$d(p,q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2}. \tag{10}$$

When we look at the speedups they have to be adapted to deal with the extra dimension.

Border pixels

To extract the border pixels from the object pixels, $n(p)$ from formula 7 contains two more neighbors. The object pixel is still a border pixel if formula (7) is satisfied.

Bisection plane

Instead of creating a bisection line, we now have to create a bisection plane. The location of the bisection plane is still defined in a new coordinate system with the feeding pixel p in the origin. Now the bisection plane made with any other object pixel q can be defined as

$$2q_1x + 2q_2y + 2q_3z = (q_1^2 + q_2^2 + q_3^2), \quad (11)$$

which is still halfway on the scan-line between p and q and perpendicular to this line. Pixel $b \in B$ satisfying $d(p, b) \leq d(q, b)$ has to be fed by p and pixel $b \in B$ satisfying $d(p, b) > d(q, b)$ has to be fed by q.

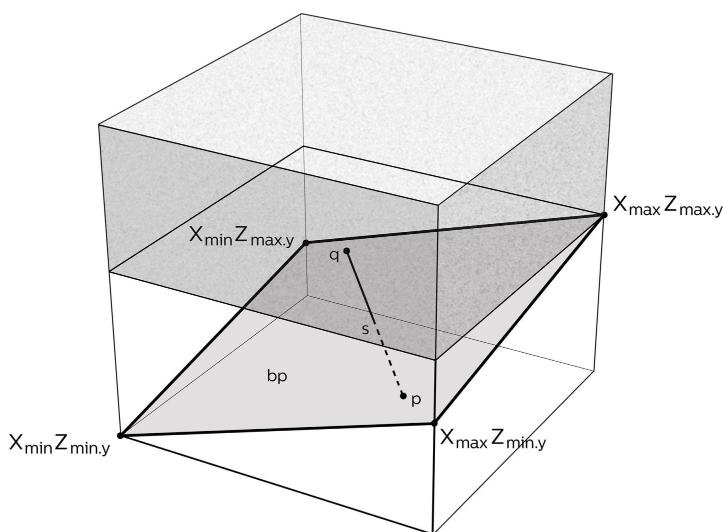

Figure 2: 3d bounding cuboid, with border pixel p, object pixel q, scan-line s and the minimum and maximum values of x and z. In this case, the y-coordinate is being updated, and the gray part is removed from the bounding cuboid based on $x_{max}z_{max}.y$.

Bounding cuboid

By combining the bisection planes we still get a convex hull, but this time in 3D space. To handle these 3D convex hulls, a rectangular cuboid can be created from the bisection planes. After a new bisection plane is created it might be possible to decrease the size of the bounding box. Figure (2) shows a bounding cuboid with scan-line s, bisection plane bp which intersect on the borders of the bounding cuboid in $x_{min}z_{min}.y, x_{min}z_{max}.y, x_{max}z_{min}.y, x_{max}z_{max}.y,$. In the 2D case, based on the direction of s, it was only needed to update two bounds, in the 3D case three bounds might need to be updated (e.g. $x_{max}, ymaxandz_{max}$) and 3 will be excluded based on the direction of the scan-line. (e.g. $x_{min}, yminandz_{min}$). In the 2D case, formula (9) is used to update the maximum x coordinate of the bounding box. In the 3D case a similar update has to be done, however the extra dimension doubles the intersection

points, changing the formula for the x-dimension into

$$bc.x_{max} = min(bc.x_{max}, max(max(y_{max}z_{min}.x, y_{min}z_{min}.x), \\ max(y_{max}z_{max}.x, y_{min}z_{max}.x). \tag{12}$$

The y and z dimensions have similar formulas. The nested maximum operator can be generalized by taking the maximum value of all the possible combinations between the three sets $\{x_{min}, x_{max}\}$ $\{y_{min}, y_{max}\}$and $\{z_{min}, z_{max}\}$ excluding the current dimension set. Note that a plane can be parallel to one of the cuboid's bounding planes. Then, no intersection can be found.

4 nd-FEED

Looping through an image in 2D (and 3D) is intuitive. Looping through an nD image is possible with n nested loops, however if n is not defined or known, a more dynamic approach is needed. To handle an nD image we first define a new coordinate system for a pixel in an image. The location of a pixel can be defined as a vector: $\vec{p} = \{p_1, p_2, \ldots, p_n\}$. Where n is the amount of dimensions and p_i the position of the coordinate in the i-th dimension. In this coordinate system all pixels $\vec{p}$ are stored in a list on position

$$L(\vec{p}) = \sum_{j=0}^{n} s_j p_j, \quad \text{with} \\ s_j = \prod_{i=j+1}^{n} v_i. \tag{13}$$

Where v is the vector with the lengths of every dimension.

The basic FEED algorithm as described in algorithm 1 does not change and keeps its basic approach. The Euclidean distance becomes the equation stated in (6).

Border pixels

Formula 7 can still be used to determine if an object pixel is also a border pixel. The amount of pixels in $n(p)$ is double the amount of dimensions.

Bisection hyper-plane

Again we make a new local coordinate system with its origin in p. We select an object pixel $q \in O$ found by the scan-line from p which can be used to define a hyper-plane between p and q. The perpendicular hyper-plane is positioned exactly on $\frac{1}{2}d(p, q)$. The hyper-plane H which is perpendicular to the scan-line between p and q is defined in nD as

$$2q_1x_1 + 2q_2x_2 + \cdots + 2q_nx_n = q_0 \\ q_0 = (q_1^2 + q_2^2 + \cdots + q_n^2), \tag{14}$$

where for each point h_i on H, $d(h_i, p) = d(h_i, q)$.

Bounding n-orthotope

Similar to the 2D and 3D cases, combining several bisection hyper-planes creates a convex hull. In an n-dimensional space a n-orthotope surrounding this convex hull makes it computational easier to handle. In n-dimensions the sets of bounds are $\{1_{max}, 1_{min}\}, \{2_{max}, 2_{min}\}, ..., \{n_{max}, n_{min}\}$, which implies the amount of combination doubles at each new dimension. Formula (12) can now be written as

$$\begin{aligned} bc.1_{max} = & min(bc.x_{max}, max(\\ & max(2_{min}.1, 3_{min}.1, \ldots, n_{min}.1), \\ & max(2_{max}.1, 3_{min}.1, \ldots, n_{min}.1) \\ & \ldots \\ & max(2_{max}.1, 3_{max}.1, \ldots, n_{max}.1). \end{aligned} \tag{15}$$

5 Theoretic Time Complexity

Basic Feed

The run-time complexity of the basic FEED algorithm has been experimental determined. Schouten and van den Broek have shown in [15], that the run-time complexity of FEED is linear in the amount of pixels in the image. They bench-marked the algorithm with the Fabbri et al. data set [7] where each image focuses on a specific image characteristic. Schouten and van den Broek created their own data set where the specific characteristics of Fabbri et al. are combined in individual images to further analyze the influences of these characteristics on the processing speed. The tests showed that the FEED algorithm is significantly faster compared to other state of the art DT algorithms.

3D FEED

Changing to 3D, has influence on run-time complexity. Determining the neighbor space for $n(x)$ in formula (7) is increased by two for the extra dimension.

More scan-lines have to be conducted for the extra dimensions and the direction of the scan-line s includes a third dimension.

More intersections have to be checked to determine if a bound of the bounding cuboid can be updated in any dimensions resulting in an increase of two extra checks. Also the maximum and minimum bounds of the third dimension need to be checked for updates.

nD FEED

Beyond 3 dimensions, the algorithm becomes increasingly slower. The neighbor space for $n(x)$ is linear in the dimensions of the images. Also more intersections have to be checked beyond 3 dimensions to determine if a bound of the bounding n-orthotope should be updated. This results in doubling the amount of checks for every new dimension: in an n-orthotope, $(2^n - 1)$ checks should be performed. This implies that the nD

FEED algorithm is less suitable for n-dimensions since it grows exponential in the amount of dimensions. Taking the exponential intersections check and the linear neighbor space into account, the time complexity becomes $\Omega(dN \times 2^d)$, d being the amount of dimensions and N being the amount of pixels.

6 Further Research

Intersecting planes

In the original FEED algorithm, intersection with other bisection lines can also lower the size of the bounding box. In 3 dimensions, the intersection of two bisection planes can be defined as a line. The maximum of this line is always on a bound. Representing the two planes as a formula on this bound can pinpoint its location and be used to update bounds. However, lines could be parallel to one or more cuboid bounds, which causes a new challenge. For n dimensions, the case is even more challenging and this needs to be further researched.

Scan-line

For the 2D case, vertical and horizontal scan-lines do not always create bisection lines with one or more intersections with all the elements in formula (9) For 3 or more dimensions there are more possibilities of parallel hyper-planes which results in non-computably values in formula (15). To deal with those specific scan-lines more general approaches should be considered. (See appendix D).

7 Conclusion

This paper introduced the n-dimensional FEED algorithm. The nD-FEED algorithm is used to calculate the DT on n-dimensional binary images. nD-FEED shows a time-complexity exponential in the amount of dimensions, in contrast to state of the art n-dimensional DT algorithms like Lucet and Felzenswalb & Huttenlocher, which are linear in the amount of dimensions. This implies that the nD-FEED algorithm will not be useful in practice for a large amount of dimensions.

8 Acknowledgments

Hereby I would like to thank dr.dr. Egon van den Broek for his extensive support during this honours-thesis. I also like to thank Gerben Aalvanger Bsc for the challenging brain storm sessions and his insight and explanation in the FEED algorithm.

References

[1] Gunilla Borgefors. Distance transformations in arbitrary dimensions. *Computer vision, graphics, and image processing*, 27(3):321–345, 1984.

[2] Gunilla Borgefors. Distance transformations in digital images. *Computer vision, graphics, and image processing*, 34(3):344–371, 1986.

[3] Thanh-Tung Cao, Ke Tang, Anis Mohamed, and Tiow-Seng Tan. Parallel banding algorithm to compute exact distance transform with the gpu. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 83–90. ACM, 2010.

[4] David Coeurjolly and Annick Montanvert. Optimal separable algorithms to compute the reverse euclidean distance transformation and discrete medial axis in arbitrary dimension. *arXiv preprint arXiv:0705.3343*, 2007.

[5] Enrique Coiras, Javier Santamarıa, and Carlos Miravet. Hexadecagonal region growing. *Pattern Recognition Letters*, 19(12):1111–1117, 1998.

[6] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and image processing*, 14(3):227–248, 1980.

[7] Ricardo Fabbri, Luciano Da F Costa, Julio C Torelli, and Odemir M Bruno. 2d euclidean distance transform algorithms: A comparative survey. *ACM Computing Surveys (CSUR)*, 40(1):2, 2008.

[8] Pedro F Felzenszwalb and Daniel P Huttenlocher. Distance transforms of sampled functions. *Theory of computing*, 8(1):415–428, 2012.

[9] Yves Lucet. New sequential exact euclidean distance transform algorithms based on convex analysis. *Image and Vision Computing*, 27(1):37–44, 2009.

[10] Calvin R Maurer Jr, Rensheng Qi, and Vijay Raghavan. A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(2):265–270, 2003.

[11] Azriel Rosenfeld and John L Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM (JACM)*, 13(4):471–494, 1966.

[12] Azriel Rosenfeld and John L Pfaltz. Distance functions on digital pictures. *Pattern recognition*, 1(1):33–61, 1968.

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

[14] Th. E. Schouten and E. L. van den Broek. Fast Exact Euclidean Distance (FEED) Transformation. In J. Kittler, M. Petrou, and M. Nixon, editors, *Proceedings of the 17th IEEE International Conference on Pattern Recognition (ICPR 2004)*, volume 3, pages 594–597, Cambridge, United Kingdom, 2004.

[15] Theo E Schouten and Egon L Van Den Broek. Fast exact euclidean distance (feed): A new class of adaptable distance transforms. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36(11):2159–2172, 2014.

[16] Frank Y Shih and Yi-Ta Wu. The efficient algorithms for achieving euclidean distance transformation. *Image Processing, IEEE Transactions on*, 13(8):1078–1091, 2004.

[17] E. L. van den Broek and Th. E. Schouten. Distance transforms: Academics versus industry. *Recent Patents on Computer Science*, 4(1), 2011.

[18] Qin-Zhong Ye. The signed euclidean distance transform and its applications. In *Pattern Recognition, 1988., 9th International Conference on*, pages 495–499. IEEE, 1988.

Appendices

A Data-set generation

For future testing an algorithm has been build to create data sets which can be used to benchmark nD algorithms including the nD-FEED algorithm. Given an object count, dimension count and a density, a random colored picture is generated.

B Binary, *O* and *B*

nD-FEED works on binary images. To handles multicolored images, a fast pass over the image has to be conducted to rewrite the image, according to formula (16, to a binary image.

$$G(p) = \begin{cases} 0 & \textit{if } I(p) \neq 0 \\ 1 & \textit{if } I(p) = 0 \end{cases} \tag{16}$$

where I is the incoming image, G the resulting image and p is the pixel being processed. Taken from G we define two sets: $O = \{p|G(p) = 0)\}$ being the set of Object pixels and $B = \{p|G(p) = 1\}$ being the set of Background pixels.

C Truncation

Background pixels which are exactly on the line between p and q needs to be fed by either one of these, not by both. According to formula (17) each pixel $b \in B$ is fed by p if $d(p, b) \leq d(b, q)$.

$$\begin{gathered} q_1x_1 + q_2x_2 + \cdots + q_nx_n = [int](\frac{((q_1^2 + q_2^2 + \cdots + q_n^2) + ud)}{2}) \\ ud = \begin{cases} 0 & \textit{if } q_j \geq 0 \\ 1 & \textit{otherwise} \end{cases} \end{gathered} \tag{17}$$

After truncation to an integer of H it is possible a pixel p which satisfies $d(p, b) = d(p, q)$ will no longer be fed by b. An adaptation ud is needed for coordinates that are on the bisection hyper-plane H to ensure b is fed by p.

D Parallel bisection

In section BLA we found that a bisection plane can be parallel to a bound of the bounding cuboid. These bisection planes are based on a scan-line containing a 0 in all but one dimension. Parallel planes will give errors when formula BLA is checked since there is no minimum or maximum value for the dimension containing a 0. For this reason one might limit the values of scan-line s to $s_i \in \mathbb{Z}_0^+$.

A second option to handle this problem is to conduct a preliminary check on the scan-line to see if all dimensions contains a 0 except for one. If this is the case, the scan-line can be used to create the initial bound.