A Novel Approach to Efficient Distance Map Calculation Using Straight Wall Optimization

Kari Väkevä, Espoo, Finland

Abstract

Traditional distance map computations rely on exhaustive pixel-wise searching, leading to performance bottlenecks, especially in high-resolution images. In this paper, we present an exact algorithm exploiting the pixel-grid's straight lines to achieve 1000 x speedup over the naïve search.

Keywords

Distance Transfer, Distance map, Euclidean distance, Exact method, Algorithm optimization.

1. Introduction

Computing distance maps efficiently has long been a challenge in image processing, particularly when dealing with large image resolutions. Conventional methods, such as naive per-pixel searching, exhibit quadratic complexity, making them infeasible for real-time applications. The state-of-the-art techniques, including Euclidean Distance Transforms (EDT) [1] and Multi-Source BFS (MS-BFS) [2], provide optimizations but may still suffer from excessive computational overhead in complex scenarios.

Here, we propose a new method that leverages the geometry of structured edges, referred to as the "Straight Wall Effect." This technique enables targeted distance calculations in predictable directions, reducing the computational burden from $O(n^2)$ complexity to $O(n)$ in specific cases, particularly for structured shapes such as rectangles and aligned segments.

2. Methodology

2.1 Conventional Approaches

In naive methods, each background pixel iterates outward to find the nearest edge pixel (see Fig.1a). This approach guarantees correctness but results in excessive computations. Traditional distance transform algorithms, such as Felzenszwalb and Huttenlocher's EDT [3], mitigate this by precomputing nearest edges through one-dimensional sweeps, leveraging lower-envelope functions. See Fig.1b for the basic EDT operation.

2.2 Straight Wall Optimization

Our method takes advantage of the fact that, in structured images where edges form "walls," the closest point to any background pixel along a perpendicular direction to the wall can be found deterministically. Instead of performing a general distance search across multiple directions, our optimized approach calculates distances only for pixels lying on specific linear paths perpendicular to structured walls.

Although digital outlines are never perfectly straight in continuous space, pixel-level rows, columns, and diagonals are. And, with an analysis of closest neighbor-pixels, it is easy to decide which operation to apply to each outline pixel, as explained in Figure 1c.

The implementation follows these steps:

1. **Edge Detection:** Identify structured black regions (walls).
2. **Direction-Based Distance Propagation:** Assign distance calculations only to linear pixel paths perpendicular to the nearest edge. This eliminates unnecessary calculations in areas where direct projections provide the closest points.
3. **Selective Neighbor Evaluation:** Instead of evaluating all pixels, a switch-case logic determines whether a pixel belongs to an optimal linear path.

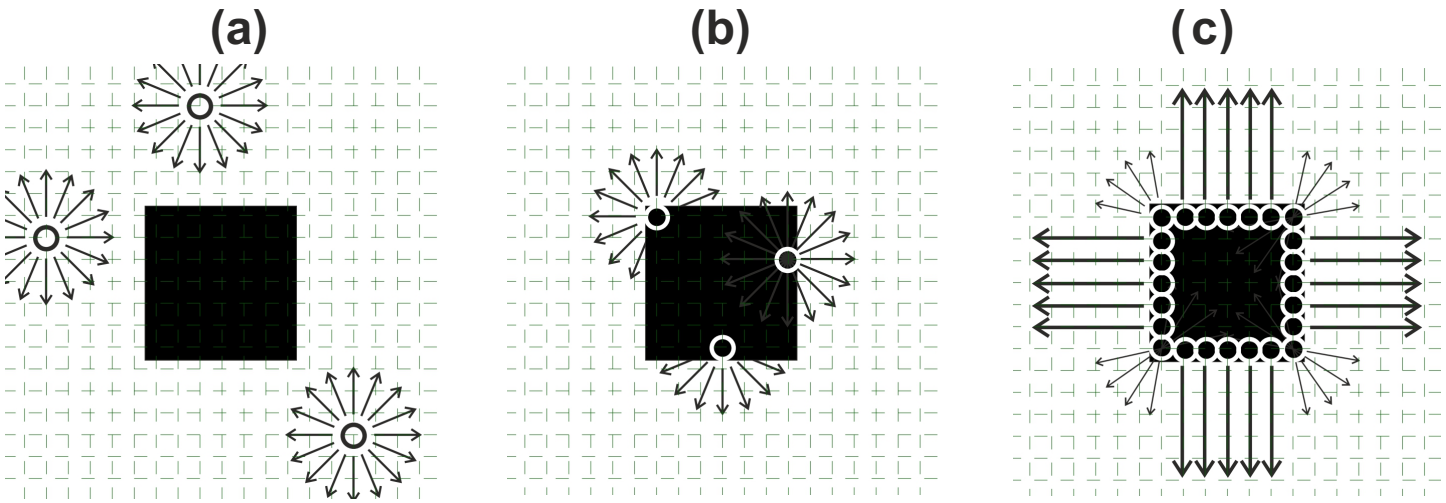
This results in a dramatically reduced number of required computations compared to conventional exhaustive distance searches.

3. Results

We tested our approach on a 1000 × 1000 image containing structured edge regions. Three implementations were compared:

 First version created 4 Jun 2025; Updated: 9 Jun 2025

Figure 1 depicts the operation of the evaluated methods in a simplified way in a reduced image scale. The naive method (a) seeks after the closest black pixel, separately for each white pixel. This gets increasingly costly when the image resolution expands. However, the basic EDT method (b) turns the search the other way round, and calculates a distance map by starting from each outline pixel of all black objects, and updates the map every time when finding a new minimum distance for a certain white pixel. This is more efficient because there are now less starting-points for the search than in the previous method. Furthermore, the number of starting-points increases linearly when the image gets larger. The Straight-wall method (c) analyzes the black object's outlines on pixel-level. Instead of performing a general distance search across multiple directions, our optimized approach calculates distances only for pixels lying on specific linear paths perpendicular to structured walls. The corner-pixels of the black object need to directionally seek corners of the image sector-wise, but nevertheless the speed of the linear-path calculations dominate the overall performance.



1. **Naive pixel-wise searching:** 809 seconds.
2. **Basic EDT-based method:** 41 seconds.
3. **Straight Wall Optimization:** <1 seconds on a 3 GHz desktop.

Our method achieves orders of magnitude improvement in speed without sacrificing precision. As the resolution increases, performance gains scale linearly, making this approach particularly suitable for real-time applications.

4. Discussion

The Straight Wall Effect optimization fundamentally transforms the way distance maps are computed in structured scenarios. While it provides exceptional speed gains for images containing aligned edges, the approach may require additional heuristics for arbitrary shapes, such as fractals or irregular boundaries. Future work will explore adaptive heuristics to extend the method to more complex geometries.

For example, a quick statistical analysis on pixel-level patterns showed that around 75% of the outline pixels a large round object are in a wall-like straight segment, see Fig.2 for explanation about the patterns and consequent decisions. A more general analysis is for future study.

5. Conclusion

We present a novel method for optimizing distance map calculations by leveraging structured wall projections, dramatically reducing computational time. This approach demonstrates that targeted pixel evaluations can replace exhaustive searches, achieving substantial efficiency improvements. Our findings suggest broad applicability in real-time graphics processing, robotics, and medical imaging applications.

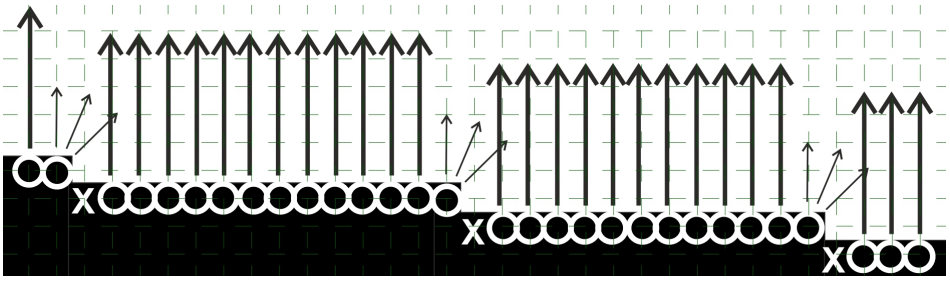
References

[1] "Distance transform", in wikipedia english. https://en.wikipedia.org/wiki/Distance_transform

[2] Devansh blog. "Multisource BFS for your FAANG coding interviews". https://medium.com/geekculture/multisource-bfs-for-your-faang-coding-interviews-d5177753f507

[3] Felzenszwalb, Pedro F., and Daniel P. Huttenlocher. "Distance Transforms of Sampled Functions". THEORY OF COMPUTING, Volume 8 (2012), pp. 415–428. www.theoryofcomputing.org

Figure 2 shows an example of other black-object shapes. Here is a detail of a round objects outline. As we can see, when going into pixel level, there are many straight-wall like sections to be fully optimized. There are three "corner"-pixels, that need a directed sector-wise threatment. However, there are also three "inner"-pixels (marked as "X") that could be entirely skipped, because they do not yield new minimums into the map; which is easy to add into our methods implementation at no extra cost.



Appendices

Appendix 1

Detailed Operation of the Optimized Distance Map Computation

This appendix describes in detail the inner workings of our optimized distance map algorithm. In our implementation, we focus on reducing the number of expensive distance (Euclidean norm) calculations by exploiting the “straight wall” effect. The key idea is that when a black region (object) is surrounded by white pixels, only those pixels along the edge facing the white background (i.e. black "corner" pixels) need to be updated with precise distance values toward the white sector. The remainder of the computation may be restricted to propagation along straight lines. The following sections describe our code’s structure and logic in detail. The code is written in C++ language.

A. Data Structures and Initialization

Image & Object Representation

- Image Matrix:
A two-dimensional Boolean array `image[ImgSz][ImgSz]` represents the image. A pixel value of `false` means white (background), while `true` represents a black pixel (object). In our experiment, the image is 1000×1000 pixels.

- Object Placement:
A black square is drawn in the center. The square’s size is set to one–third of the total image dimension. The starting and ending indices (`ObjBegin` and `ObjEnd`) are calculated so that the object is centered.

Distance Table

- Distance Map (distTbl):
A corresponding 2D array of double-precision values is used to store the computed distances. Initially, every pixel is assigned an “infinite” distance (set using a constant `Inf = 1.0e+099`).

Neighbor Data

- Neighborhood Array (`nb`) and Combination Variable (`nbComb`):
To detect edge pixels reliably, we inspect immediate 8-neighborhood values. In the current implementation, only four cardinal neighbors (up, left, right, down) are explicitly used. Their Boolean values are stored in an array `nb[8]` (even though only indices 1, 3, 4, and 6 are used) and then combined into a bit-code (`nbComb`) that governs the subsequent control flow.

B. Fundamental Distance Functions

1. Pixel Comparison – `Compare2`
The function

```
bool Compare2(int x, int y)
```

checks whether a given coordinate lies within the image bounds. If it does, the function returns the negation of `image[x][y]` (i.e., `true` if the pixel is white and `false` if black). This is central for our update routines because we wish to update the distance map

only for background (white) pixels adjacent to the object.

2. Euclidean Norm Calculation – `CalcHypotenuse2`
The function

double CalcHypotenuse2(int len1, int len2)

computes the Euclidean distance using the formula:

$$s = \sqrt{(len1^2 + len2^2)}$$

This function is invoked repeatedly when a candidate white pixel's distance from a black (object edge) pixel is being evaluated.

C. Update Functions for Distance Propagation

Our implementation employs several specialized update routines that propagate a distance value along a single row or column based on the "straight wall" observation:

1. General Update – `UpdateDistanceMap`

This function is intended as the fallback update routine when the neighboring conditions do not match an optimized direction. It uses an incremental layer approach (indexed by `i`) to examine pixels in a square "shell" centered at a given black pixel at position (x, y). Inside the helper function `IsMatch2`, the code:
- Iterates horizontally from (x-i, y) to (x+i, y) for both the top and bottom edges of the square.
- Iterates vertically from (x, y-i+1) to (x, y+i-1) for the left and right edges.
- For every candidate white pixel (detected via `Compare2`), the distance is updated with:

distTbl[.] = min ({current value}, CalcHypotenuse2(...))

There are also further refinements (e.g., backtracking criteria) to terminate the search once the optimal or sufficiently low distance is achieved.

2. Directional Update Functions

For pixels that reside on an object's "straight wall," only one-dimensional propagation is necessary. Based on the values stored in the neighbor bit array, the algorithm distinguishes four specific cases:
- **Row-Leftward – `UpdateDistanceMap3`:**
  For a given black pixel (x, y) where the leftward neighbor is white, the function iterates with increasing I (i.e., moving left) and updates:

distTbl[x-i][y] = min (distTbl[x-i][y], i)

- **Row-Rightward – `UpdateDistanceMap4`:**
  Similar to the leftward version, but propagation is to the right:

distTbl[x+i][y] = min (distTbl[x+i][y], i)

- **Column-Upward – `UpdateDistanceMap5`:**
  Propagation upward along the column:

distTbl[x][y+i] = min (distTbl[x][y+i], i)

- **Column-Downward – `UpdateDistanceMap6`:**
  Propagation downward:

distTbl[x][y-i] = min (distTbl[x][y-i], i)

Each of these directional functions solely updates pixels along a single axis. This is the key "straight wall" optimization: by restricting the update direction, we avoid the quadratic number of calculations required when considering all directions.

D. Overall Workflow

1. Initialization
- The distance map is initialized so that every pixel holds an "infinite" distance.
- A black square (object) is drawn in the center.

2. Identification of Edge Pixels
- The algorithm iterates over all pixels.
- For every pixel that is black (i.e., part of the object), the code checks its immediate 4–neighbors. If any neighbor is white (using:

if(!image[x-1][y] || !image[x+1][y] || !image[x][y-1] || !image[x][y+1])

), then the pixel is flagged as being on the boundary.

3. Conditional Updating via Switch-Case
- Based on the state of the four immediate neighbors, a bit-combination (`nbComb`) is created.
- Using a `switch-case` structure, the algorithm selects one of the optimized Update routines:
  - Cases corresponding to specific bit patterns (e.g., `0x0d`, `0x0b`, `0x07`, `0x0e`) trigger the one-dimensional propagation functions (i.e., UpdateDistanceMap3–6).

- For other configurations, the general `UpdateDistanceMap` is invoked.

This conditional branching ensures that when the object boundary aligns in a straight line relative to the background, only the most efficient directional update is triggered.

*NB: For simplicity, this demo uses 4-neighbors, but the full implementation **must** encode all 8-neighbor values in the nbComb structure to detect all cases and apply a correct function!*

4. Time Measurement and Debug Output
- The total runtime is measured and printed.
- Debug data (selected rows and columns from the distance map) is output to verify correctness.

E. Discussion and Heuristics

The core idea behind the optimization is that for many structured objects (e.g., a centered square), only a small subset of the object's edge pixels actually govern the minimal distances to the background. By using a combination of neighbor evaluation and directional updates:
- The overall number of distance (Euclidean norm) calculations is reduced dramatically.
- The propagation along the "straight wall" is linear—yielding an O(n) behavior along that axis—versus a full two-dimensional search that might be $O(n^2)$.

This effect, which we term the "Straight Wall Effect," is further exploited by the use of a `switch-case` mechanism. The bit-combination of neighbor values allows the algorithm to rapidly decide which single-axis update is most appropriate, avoiding unnecessary iterations.

F. Limitations and Future Refinements

While the current implementation demonstrates impressive speedups, especially on large images, some aspects remain experimental:
- The system currently covers only the four primary directions. Robust handling of diagonal and irregular boundaries require additional cases.
- Further validation on more complex images (beyond the structured black square) is needed to generalize the proposed heuristic.

Conclusion

This appendix has detailed the operation of our optimized distance map computation. By focusing on "straight wall" optimization and conditional directional propagation, the algorithm reduces computational overhead significantly over the naive method. This approach is particularly powerful for images with structured object boundaries, as demonstrated by the substantial performance improvements observed in our experiments.

Appendix 2.

The code (see the following pages).

```
//
// main_TEST.cpp
//

#include <stdio.h>
#include <math.h>
#include <time.h>

const int ImgSz = 1000;///TOT size of the image dimension
bool image[ImgSz][ImgSz] = { 0 };//initialize all "white"

const int ObjSz = ImgSz/3;//black square dimension
const int ObjBegin = (ImgSz - ObjSz)/2;//front edge location of the black square
const int ObjEnd = ObjBegin + ObjSz;//rear edge location of the black square

const int w = ImgSz, h = ImgSz;

double distTbl[ImgSz][ImgSz];
const double Inf = 1.0e+099;

bool nb[8] = { 0 };//8-neighbour pixels
int nbComb = 0;//bit-combined value

bool z = 0;//*** Backtracking (if zero) ***

inline double Min(double a, double b) {
	if (a<b)
	{
		return a;
	}
	else
	{
		z = 1;
		return b;
	}
}


bool Compare2(int x, int y)
{
	if (x < 0 || x >= w || y < 0 || y >= h) return false;//(stop if outside)
	else return !image[x][y];
}
double CalcHypotenuse2(int len1, int len2)
{
	return pow(len1*len1 + len2*len2, .5);
}
bool IsMatch2(int x, int y, int i)
{
	for (int a = x - i; a <= (x + i); a++)
	{
		if (Compare2(a, y - i))
			distTbl[a][y-i] = Min(distTbl[a][y-i], CalcHypotenuse2(a - x, i));
		if (Compare2(a, y + i))
			distTbl[a][y+i] = Min(distTbl[a][y+i], CalcHypotenuse2(a - x, i));
	}
	for (int b = y - i + 1; b < (y + i); b++)
	{
		if (Compare2(x - i, b))
			distTbl[x-i][b] = Min(distTbl[x-i][b], CalcHypotenuse2(i, b - y));
		if (Compare2(x + i, b))
			distTbl[x+i][b] = Min(distTbl[x+i][b], CalcHypotenuse2(i, b - y));
	}
	if (i==0) return true;
	else return z;
}
```

```
void UpdateDistanceMap(int x, int y)
{
	int LimA = w;
	for (int i = 0; i < LimA; i++)
	{
		z = 0;
		if (IsMatch2(x, y, i)==0)
		{
			return;//*** Backtracking ***
		}
	}
}
//
//Optimized versions of the Map
//
void UpdateDistanceMap3(int x, int y)//Row-Leftward version
{
	int LimA = w;
	for (int i = 0; i < LimA; i++)
	{
		z = 0;
		if (Compare2(x - i, y)) distTbl[x-i][y] = Min(distTbl[x-i][y], double(i));
		if (i && !z) return;//*** Backtracking ***
	}
}
void UpdateDistanceMap4(int x, int y)//Row-Rightward version
{
	int LimA = w;
	for (int i = 0; i < LimA; i++)
	{
		z = 0;
		if (Compare2(x + i, y)) distTbl[x+i][y] = Min(distTbl[x+i][y], double(i));
		if (i && !z) return;//*** Backtracking ***
	}
}
void UpdateDistanceMap5(int x, int y)//Column-Upward
{
	int LimA = w;
	for (int i = 0; i < LimA; i++)
	{
		z = 0;
		if (Compare2(x, y+i)) distTbl[x][y+i] = Min(distTbl[x][y+i], double(i));
		if (i && !z) return;//*** Backtracking ***
	}
}
void UpdateDistanceMap6(int x, int y)//Column-Downward
{
	int LimA = w;
	for (int i = 0; i < LimA; i++)
	{
		z = 0;
		if (Compare2(x, y-i)) distTbl[x][y-i] = Min(distTbl[x][y-i], double(i));
		if (i && !z) return;//*** Backtracking ***
	}
}



int main() {
	{
		const time_t startT = time(0);

		for (int i = 0; i < ImgSz; i++)
		{
			for (int j = 0; j < ImgSz; j++)
			{
				distTbl[i][j] = Inf;//initial distance set to "infinity"
```

```
                }
            }
            for (int i = ObjBegin; i < ObjEnd; i++)
            {
                for (int j = ObjBegin; j < ObjEnd; j++)
                    image[i][j] = 1;//set "black"
            }
            for (int x = 0; x < ImgSz; x++)
            {
                for (int y = 0; y < ImgSz; y++)
                {
                    if (image[x][y])
                    {
                        nb[1] = image[x  ][y-1];
                        nb[3] = image[x-1][y];
                        nb[4] = image[x+1][y];
                        nb[6] = image[x  ][y+1];
                        if (!nb[1] || !nb[3] || !nb[4] || !nb[6])
                        {
/*                          nb[0] = image[x-1][y-1];//*** TODO ***
                            nb[2] = image[x+1][y-1];
                            nb[5] = image[x-1][y+1];
                            nb[7] = image[x+1][y+1];*/
                            nbComb = nb[1] + nb[3]*2 + nb[4]*4 + nb[6]*8;
                            switch (nbComb)
                            {
                            case 0x0d:
                                UpdateDistanceMap3(x, y);//Row-Leftward version
                                break;
                            case 0x0b:
                                UpdateDistanceMap4(x, y);//Row-Rightward version
                                break;
                            case 0x07:
                                UpdateDistanceMap5(x, y);//Column-Upward
                                break;
                            case 0x0e:
                                UpdateDistanceMap6(x, y);//Column-Downward
                                break;
                            /*  TODO: add here more optimizations cases  */
                            default:
                                UpdateDistanceMap(x, y);
                            }
                        }
                    }
                }
                if ((x%10)==0) printf(".");
            }
            const time_t stopT = time(0);

            printf("\nready (dur: %d)\n", int(stopT-startT));

            printf("\n\nDEBUG DATA PRINT:\n\n");
            for (int i = 0; i < ImgSz; i++) printf(" %f %f %f %f\n", distTbl[i][0], distTbl[i][ImgSz-1], distTbl[0][i], distTbl[ImgSz-1][i]);
        }
END:
        _fcloseall();
        getchar();//"pause"
        return 0;
}
```