Asymptotic analysis applied to skeletonization by distance algorithms

Fabrizio Franco
School of Computer Science
University of Engineering and Technology UTEC
Email: fabrizio.franco@utec.edu.pe

Carlos Reátegui
School of Computer Science
University of Engineering and Technology UTEC
Email: carlos.reategui@utec.edu.pe

Christian Ledgard
School of Computer Science
University of Engineering and Technology UTEC
Email: christian.ledgard@utec.edu.pe

Maor Roizman Gheiler
School of Computer Science
University of Engineering and Technology UTEC
Email: maor.roizman@utec.edu.pe

***Abstract*—This article explains the background behind the Skeletonization by distance method. Furthermore, it analyzes theoretically and empirically different algorithms for the calculation of the distance transform map, as well as it makes a theoretical comparison between two algorithms for skeletonization, one of them designed by ourselves for this research.**

***Index Terms*—Skeletonization, Euclidean Distance, Skeleton Strength Map.**

I. Introduction

The skeletonization process, also known as the process of obtaining the medial axis, corresponds to an important process to be able to determine the minimum relevant structure for an object that, despite eliminating fragments of it, manages to maintain its main properties morphologically and topologically. This process can be carried out by different methods. Section III gives a general overview of multiple approaches, like thinning algorithms, discrete domain algorithms based on the Voronoi diagram, algorithms based on mathematical morphology, and the approach used in this work, distance transform base algorithms [1]. A two dimensions skeleton contains mainly three types of points: end-points with one neighboring point in the skeleton, common points with one neighbor in each side (two neighbors in total), and branch-points with more than two neighbors. In the following article we will develop the skeletonization method by distance using methods with branch-points, exactly 8 locations per point, and a detailed asymptotic analysis of algorithms will be carried out.

II. Applications

A. Object representation

The skeletonization process can be used to represent an object in a minimal way. This process can be carried out in both two and three dimensions. It allows to fulfill morphological analysis of images and topological, for the study of surfaces. [2]

B. Medical imaging

Since skeletonization gives morphological characteristics of 2D or/and 3D images, it is useful for the analysis and recognition of images applied in biological tubular structures. From the characteristics of the skeleton, it can detect, for example, spinal stenosis or determine characteristics that are related to the patient's health condition. [2]

III. Related Works

A. Skeletonization by thinning

'Thinning is an image processing operation in which binary valued image regions are reduced to line that approximate the center skeletons of the regions' [3]. This method is considered a morphological technique that uses a substructure commonly known as 'Structuring Element' (SE). This substructured is positioned at all possible locations in the image and it is compared with the corresponding neighborhood of pixels [3].

B. Skeletonization based on mathematical morphology

This approach proposes both an arithmetic and a geometric way to find the skeleton of a binary image. 'Blum considered two principal approaches to find the medial axis. First, he used the "symmetric point distance" from a skeleton point to the boundry. Second, he showed that the symmetric axis is the place of the centers of the "maximal disks" inscribable inside a filled-in image object' [4].

IV. Experiments and Results

A. Skeletonization by distance

Two distance transform algorithms were implemented to be compared.

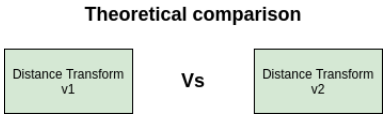

Fig. 1. Theoretical comparison

1) First approach: The first algorithm approaches the distance transform as a shortest path problem, where the graph is built from the image [5].

It receives as input an image and a distance function.

Each pixel is a node of the graph, and each pixel has at most eight neighbors, and an edge to each neighbor. Furthermore, each pixel contains a cost assigned to it, which represents its distance to the closest border pixel.

Some variables must be initialized for the algorithm to work properly (see 1). The variable called `distances` holds the distance of each pixel to the closest border pixel; the variable called `borders` holds the closest border pixel to each pixel and `pixels_queue` is a priority queue that will be used to propagate from the border pixel to all the pixels of the image in order to find the closest distance of each pixel to a border pixel.

```
1  distances = empty
2  borders = empty
3
4  pixels_queue = initialize-pq()
5
6  for each pixel in pixels:
7      if pixel is border pixel:
8          distances(pixel) = 0
9          border(pixel) = pixel
10         pixels_queue.insert(pixel)
```

Listing 1. Initialize variables for distance transform algorithm 1

Once those variables have been initialized, the algorithm is ready to calculate the distance transform (see 3). For each neighbor in the pixel queue, the distance between the root of the pixel and the neighbor is calculated. If this distance is smaller than the current distance of the neighbor, the distance and the root of the neighbor are updated. Finally, if the pixel is not in the queue, add it to propagate from it.

```
1  while pixels_queue has elements:
2      pixel = pixels_queue.pop
3      for neighbor in eight_neighbors(pixel):
4          distance = distance between borders(p) and
       neighbor
5          if distance < distances(neighbor):
6              distances(p) = distance
7              borders(neighbor) = borders(pixel)
8              if neighbor is not in pixels_queue:
9                  insert neighbor to pixels_queue
```

Listing 2. Calculate distance transform using algorithm 1

Let n be the number of pixels the input image and $T(n)$ be the execution time of this algorithm.

Clearly, the initialization of variables from listing 1 takes linear time, since this part of the algorithm iterates once through each pixel.

Now, let us analyze listing 3 from the inside out. An insertion is made to the pixels queue in line 9; since the pixel queue is constructed as a priority queue (heap) and can have at most n elements, this operation takes $O(lg(n))$. The `for` loop of line 3 takes at most 8 iterations, since each pixel has at most 8 neighbors; the `for` loop executes an $O(lg(n))$ operation at most 8 times, and for that reason it takes $O(lg(n))$. The `while` loop of line 1, executes while the pixel queue is not empty. Since line 8 makes sure no duplicated pixels can exist in the `pixels_queue`, and each pixel is neighbor to at most 8 other pixels, the total running time of the algorithm is $T(n) = O(nlg(n))$.

2) Second approach: This alternative approach, proposed by [6], finds the distance of each pixel from their neighbors. It uses 2 passes to analyze the image: a top-down, left-right and a bottom-up, right-left.

```
1  firstPassDistance()
2    for i = 0 to rows:
3      for j = 0 to columns:
4        if image[i][j] > 0:
5          loadNeighborTop(i,j) //Load the top three
       neighbors of the pixel.
6          image[i][j] = min() + 1
7
8  secondPassDistance()
9    for i = rows to 0:
10     for j = columns to 0:
11       if image[i][j] > 0:
12         loadNeighborBottom(i,j) //Load the bottom
       three neighbors of the pixel.
13         image[i][j] = minSecond()
14         if image[i][j] < newMinValue:
15           newMinValue = image[i][j]
16         if image[i][j] > newMaxValue:
17           newMaxValue = image[i][j]
```

Listing 3. Calculate distance transform using algorithm 2

Let n be the total number of pixels of an image, and $T(n)$ be the running time of the algorithm. Both passes iterate through each pixel of the image. Thus, $T(n) = O(n)$.

3) Generation of Images and Matrix's: In order to generate a matrix from an image, and vice-versa, we prepared a script in Python 3.8. using the library's `imageio` and `numpy`. Additionally, we use the `ImageJ 1.53a` in order to generate our fourth image, subtracting the background.

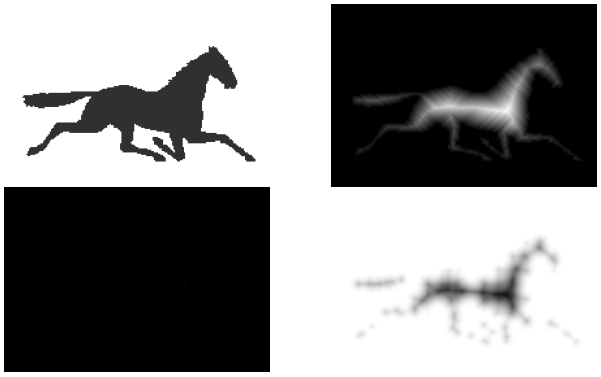

Fig. 2. Plotting Horse

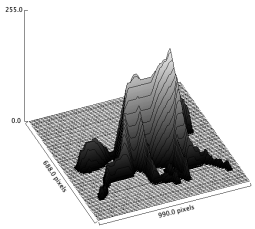

Fig. 3. Surface Plot of the Distance Transform (Horse)

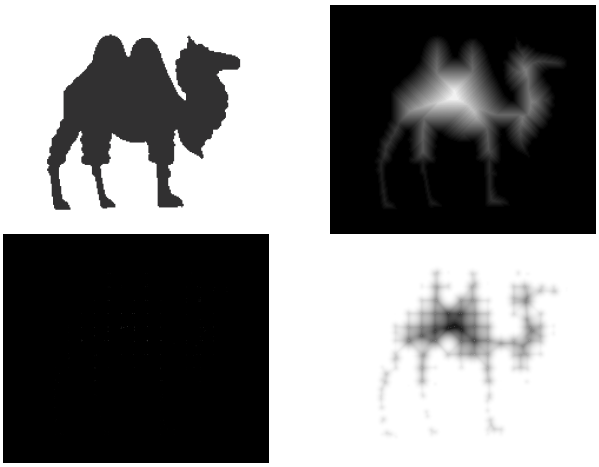

Fig. 4. Plotting Camel

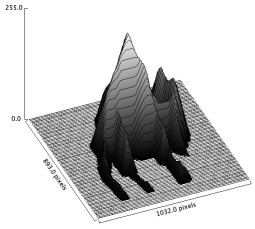

Fig. 5. Surface Plot of the Distance Transform (Camel)

4) Results: In order to test the execution time of the algorithms, we run 3 experiments on 3 different image sizes. We use the library *chrono* to measure exact time in *milliseconds*.

After analyzing both algorithms, the Second Approach has an overall better performance on running time using four neighbors. On the other hand, the first approach uses eight neighbors, and it can run multiple distances (Manhattan, Euclidean, Chebyshev) but has a significantly higher running time.

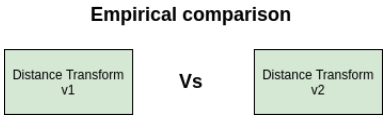

Fig. 6. Empiric comparison

Experiment	ImageSize (px)	First approach (ms)	Second approach (ms)
1	495x344	1642	240
2	495x344	1611	224
3	495x344	1626	232
4	495x344	1729	224
5	495x344	1718	219
1	990x688	8048	854
2	990x688	8822	767
3	990x688	8094	879
4	990x688	8390	812
5	990x688	8194	819
1	1980x1376	50084	3179
2	1980x1376	48165	3293
3	1980x1376	48282	3349
4	1980x1376	49365	3197
5	1980x1376	49303	3392

TABLE I
EXPERIMENTATION RESULTS

ImageSize (px)	AVG of First approach (ms)	AVG of Second approach (ms)
495x344	1665,2	227,8
990x688	8309,6	826,2
1980x1376	49039,8	3282

TABLE II
EXPERIMENTATION SUMMARY

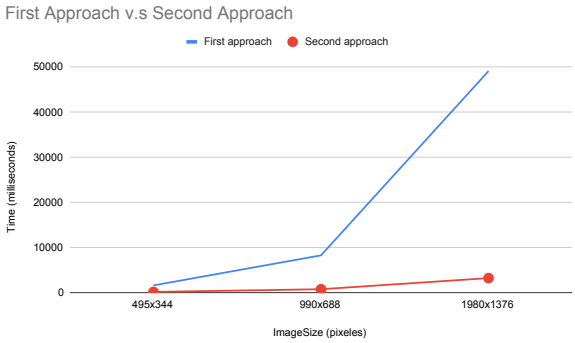

Fig. 7. Comparison of time execution

B. Skeletonization by Distance and Strenght Map

"The basic idea of the SSM approach is to select critical points from the Skeleton Strength Map, named SSM, and connect them by geodesic paths computed with Dijkstra's shortest path algorithm" [7].

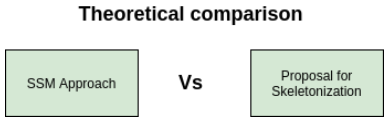

Fig. 8. Empiric comparison

The principal steps for this approach are:

1) Distance transform: Get from the original image the distance transform.
2) Skeleton Strength Map: Calculate the SSM using the Gradient Vector Field.

3) Local Maxima: Obtain the points that satisfy $SSM(Image) \geq maxSSM(Image')$ [1].
4) Critical points selection: Obtain the points that are most likely to be part of the skeleton based on the magnitude of its gradient.
5) Critical points connection: The result of this process is the final Skeleton.

```
GradientVectorField:
    gradient = []
    for pixel in pixels:
        gradient[pixel] = CalculateGradient(pixel)

FindSSM:
    SSM = {}
    for pixel in pixels:
        add to SSM max(0, sumOf8NeighborsGradients(
    pixel))
    returun SSM

ConnectedComponents:
  SSM = FindSSM()
  connectedComponents = {}
  for pixel in pixels:
    maxSSMofNeighbors = maxSSMofNeighbors(pixel)
    if SSM(pixel) >= maxSSMofNeighbors:
      add neighbor to region
    add pixel to connectedComponents
  return connectedComponents

CriticalPointsSelection:
  connectedComponents = ConnectedComponents()
  criticalPoints = {}
  for connectedComponent in connectedComponents:
    criticalPoint = connectedComponent[0].gradient
    for i = 1 to connectedComponent.size()
      if connectedComponent[i].gradient <
    criticalPoint.gradient:
        criticalPoint = connectedComponent[i]
    add criticalPoint to criticalPoints
    add connectedComponent.endPoints to
    criticalPoints
  return criticalPoints

CriticalPointsConnection:
  criticalPoints = CriticalPointsSelection()
  gradientPaths = getGradientPaths()
  gradientPathsEndpoints = gradientPaths.endpoints
  maxPoint = getMaxDistanceTransform()
  skeleton = graph with criticalPoints and
    gradientPathsEndpoints as nodes
  for node in graph
    dijkstra(maxPoint, node)
  return skeleton
```

Listing 4. Skeleton Strength Map (SSM) pseudocode

Clearly, the functions `GradientVectorField` and `FindSSM` take linear time, since they iterate through all the pixels of the image and perform constant time operations.

The function `ConnectedComponents` iterates through all the pixels, and for each pixel it iterates through its 8 neighbors. Thus, it also takes linear time.

The function `CriticalPointsSelection` takes $O(m * r)$, where m is the number of connected components, and r the maximum number of points in a given component.

The function `CriticalPointsConnection` takes $O(p * ((p + e) * lg(p))$, where $p = |criticalPoints| + |gradientPaths|$ and e is the total number of edges generated by the graph.

Thus, the running time of the algorithm is dominated by $T(p) = O(p * ((p + e) * lg(p))$.

C. Skeletonization by Proposal approach

To skeletonize the distance transforms we chose our own approach. First, we fragment the image into 'S' segments. Then we implement a function to get the top 'N' pixels with greater intensity. From this, we were able to find the most relevant points regarding each segment and thus the most relevant for the generation of the skeleton. An improvement we implement and analyze was calculating the average of the values and returning the top 75% of the results in the segments. With this improvement, we thought that we would make the more critical segments noticed. However, the results were not as expected because it carries irrelevant pixels to our result. The complexity of the proposed algorithm is $n^2 \log(n)$ because the number of segments is n/c, where $c > 0$, and we iterate through all the quadrants and sort them in order to get the maximum local points.

```
FragmentImage(distanceTransform, rationNumFragments)
     //Get image segments
    numElementX = distanceTransform[0].size()/
    rationNumFragments
    numElementY = distanceTransform.size()/
    rationNumFragments
    for x = 0 to numElementosy:
        for y = 0 to numElementosX:
            new segment(x,y)

    a, b = 0
    for x = 0 to distanceTransform.size():
        pixelesVisitedX++
        if pixelesVisitedX >= totalPixelesX: a++
    pixelesVisitedX = 0
        for y = 0 to distanceTransform[0].size():
            pixelesVisitedY++
            if pixelesVisitedY >= totalPixelesY: b++
     pixelesVisitedY = 0
            segment[a][b].segment.pushBack(x,y,
    distanceTransform[x][y])

GetTopNStrongestPoints(numStrongestPixel) //Local
    maxima
    for segment in segments:
        segment.sort()
        for strongestPixel = 0 to numStrongestPixels
    :
            pixel = fragment[strongestPixel]
            finalMatrix[pixel.x][pixel.y] = pixel.
    intesinty
```

Listing 5. Skeletonization method proposal

Check the implementation *here*.

```
MatchDots(DistanceTransformMap DSM):
    dots = getDotsFromDTMap(DSM)
    for dot in dots:
        remove dot from dots
        dot2=nearestPoint(dot,dots)
        generateEdges(dot,dot2)



// UTILITY FUNCTIONS
```

```
nearestPoint(dot,dots):
    nearest=dots[0]
    size = size of dots - 1
    for p=1 to size:
        if(distance(dot,dots[p]) < distance(dot,
    nearest) and not edge between dots[p],dot)
            nearest = dots[p]
    return nearest

// MatchDots function create an edge between two
    dots (vertices)
```

Listing 6. Joining dots of Skeleton

V. Conclusions

- Within the different distance transform algorithms there are some that allow greater morphological precision such as SSM, which guarantees the connection of the critical points with geodesic paths. On the other hand, there exist other skeletonization algorithms that give a fairly precise result, but do not guarantee connectivity between all the sections of the skeleton.
- In theoretical analysis, the first approach takes 8 neighbors, and it is design to be generalized into various types of distances (Manhattan, Euler, Chebyshev) taking $T(n) = O(nlg(n))$ complexity. The second approach only uses 4 neighbors and it is optimized for one type of distance (Chebyshev) taking $T(n) = O(n)$ time complexity.
- In the empirical analysis, theoretical complexity times were successfully validated, being consistent with expectations. The first approach grows in a larger rate than the second approach.
- Both approaches used for skeletonization can not be used for 3D images, since the algorithms are not able to recognize patterns in variable dimensions, for example, if we take the X and Y axis, how might we find patterns between (X and Z) or (Y and Z)? Can we apply the analysis for each pair of axis? It is not possible to do that without modifications since patterns are not added, they are combined. On the other hand, a new analysis would be necessary in order to calculate the new possible time complexity.
- To determine which one is asymptotically greater, we will use the definition of limits dividing both complexities, when n tends to infinity $\lim_{x\to\infty}\left(\frac{x\cdot((x+e)\cdot lg(x))}{x^2\cdot lg(x)}\right)$, in this case, the limit value is 1, that is, a constant c, for that reason $T(n)$ of the first approach is $\Theta(f(n))$ where $f(n)$ is $T(n)$ for second approach.

References

[1] X. B. L. J. Latecki, Q. Li and W. Liu, "Skeletonization using ssm of the distance transform," *International Conference on Image Processing, San Antonio, TX, 2007*, vol. 349, p. 352, 2007.

[2] B. G. . S. d. B. G. Saha, P. K., "Skeletonization and its applications – a review. skeletonization," p. 3–42, 2017.

[3] K. Kaur and D. M. Kumar, "A method for binary image thinning using gradient and watershed algorithm," *International Journal of Advanced Reasarch in Computer Science and Software Engineering*, vol. 3, 01 2013.

[4] P. Maragos and R. SCHAFER, "Morphological skeleton representation and coding binary images," *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 34, pp. 1228 – 1244, 11 1986.

[5] F. A. Z. R. A. Lotufo, A. A. Falcão, "Fast euclidean distance transform using a graph-search algorithm," *SIBGRAPI '00: Proceedings of the 13th Brazilian Symposium on Computer Graphics and Image Processing*, October 2000.

[6] T. Wu, "Distance-transformation," https://github.com/tyeonn/Distance-Transformation, 2018.

[7] X. Y. N. Cornea, D. Silver and R. Balasubramanian, "Computing hierarchical curve-skeletons of 3d objects," *The Visual Computer*, October 2005.