Zhang-Suen Fast Parallel Thinning Algorithm

NOVEMBER 2022

DIVYA NAIKEN

What is Thinning

Thinning is a process whereby the pixels of the foreground of an image is reduced to a minimal quantity where the endpoints and connectivity of all the pixels are maintained. It is easier to understand thinning pictorially using the prairie-fire analogy. Consider, an object that is set on fire, and the fire consumes the perimeter of the object and continues to burn with equal velocity inwards, where the fire's cross paths are defined as the thinned, structural representation of the original image. This outline can also be referred to as an approximation of the **medial axis**. Since the medial axis is defined as, “the set of all points having more than one closest point on the object's boundary”(Sawhney), it is not possible to obtain the exact medial axis, but rather the points that are the most stable from an estimate of the axis.

The importance of thinning lies in its ability to remove unnecessary information in images and thereby reduce process time, and computational resources. Thinning is used is many tasks, a few include, handwriting recognition, pattern recognition, and medical imaging analysis.

Theory Behind the Fast Parallel Zhang-Suen Algorithm

The Fast Parallel Algorithm is an iterative-thinning algorithm consisting of 2 sub-iterations. Each pass aims to remove specific corners of a neighbourhood of 8 pixels. In order to maintain connectivity, the first iteration deletes the northwest pixels and the second deletes the southeast pixels (Zhang & Suen, 1984).

In a neighbourhood of 8 pixels, the pixels are labeled as the following:

P9	P2	P3
P8	P1	P4
P7	P6	P5

Figure 1.0 - Points labeled in Neighbourhood

In the first sub-iteration, the following conditions are checked, if they are all met, P_1 is deleted.

a) $2 \leq B(P_1) \leq 6$
b) $A(P_1) = 1$
c) $P_2 \times P_4 \times P_6 = 0$
d) $P_4 \times P_6 \times P_8 = 0$

Where, $B(P_1)$, represents the sum of non-zero neighbours and $A(P_1)$, represents the number of pairs of points that start with a 0 and end with a 1, in the neighbourhood from P2 to P9. In the second iteration, the following conditions are checked, and if they are all met, P_1 is deleted.

a) $2 \leq B(P_1) \leq 6$
b) $A(P_1) = 1$
c) $P_2 \times P_4 \times P_8 = 0$
d) $P_2 \times P_6 \times P_8 = 0$

2 sub-iterations makes up one entire iteration, and the algorithm stops when no further points can be deleted.

As explained in the paper *A fast parallel algorithm for thinning digital patterns*, for conditions c and d, in the first iteration, the solution is that either P_4 = 0 or P_6 = 0, thus making, P_1 a north-west corner pixel Condition a preserves the endpoints and condition b keeps the points that reside between the skeleton line. A similar explanation is used for the second iteration.

Implementation

This chunk of code determines the neighbours of the pixels:

```
'''
find_neighbours : finds 8 point neighbourhood

Input:
x  - int: row index of P1
y  - int: col index of P1
IT - list: binary image

Output:
neighbourhood - list: P2-P9 in clockwise direction

P9  -  P2  - P3
P8  - *P1* - P4
P7  -  P6  - P5
'''
def find_neighbourhood(row,col,IT):
   P2 = IT[row - 1][col]
   P3 = IT[row - 1][col + 1]
   P4 = IT[row][col+1]
   P5 = IT[row + 1][col + 1]
   P6 = IT[row + 1][col]
   P7 = IT[row + 1][col - 1]
   P8 = IT[row][col - 1]
   P9 = IT[row - 1][col - 1]

   neighbourhood = [P2, P3, P4, P5, P6, P7, P8, P9]
   return neighbourhood
```

This section checks condition A:

```
'''
CONDITION A)
find_nonzero_nbs : finds the number of non_zero magnitude neighbours

Input:
neighbourhood   - list: P2-P9 in clockwise direction

Output:
n_nonzero - int: number of 1's in ordered set P2-P9
'''
def find_nonzero_nbs(neighbourhood):

   n_nonzero = sum(neighbourhood)

   return n_nonzero
```

This section checks condition B:

```
'''
CONDITION B)
find_transitions : finds the number of transitions

Input:
neighbourhood   - list: P2-P9 in clockwise direction

Output:
n_transitions   - int: number of 01 patterns in ordered set P2-P9
'''
def find_transitions(neighbourhood):
   nd = neighbourhood  # for simplicity
   n_transitions = 0
   prev = 0
   curr = 0

   for i in range(len(nd)):
       curr = nd[i]

       if prev == 0 and curr == 1:
           n_transitions += 1
       prev = curr

   return n_transitions
```

This section checks conditions C and D for **both** iterations

```
'''
ITER 1 & 2 CONDITION C and D)
find_product1 : finds the product of P2, P4, P6  and P4, P6, P8 for sub-iteration 1
               or, P2, P4, P8, and P2, P4, P8 for sub-iteration 2

Input:
neighbourhood       - list: P2-P9 in clockwise direction
subiter             - int: specifies which iteration the algorithm is on

Output:
product1, product 2 - int, int:  product of North, East and South points,
                                products of the East, South and West points
'''
def find_product(neighbourhood, subiter):

   if subiter == 1:
       product1 = neighbourhood[0] * neighbourhood[2] * neighbourhood[4]
       product2 = neighbourhood[2] * neighbourhood[4] * neighbourhood[6]
   elif subiter == 2:
       product1 = neighbourhood[0] * neighbourhood[2] * neighbourhood[6]
       product2 = neighbourhood[0] * neighbourhood[4] * neighbourhood[6]

   return product1, product2
```

This section deletes pixels that have satisfied the 4 conditions for either sub-itereation

```
def delete_p(to_be_deleted, img):
   for pair in to_be_deleted:
       x = pair[0]
       y = pair[1]

       img[x,y] = 0

   return img
```

This is the first sub iteration

```
'''
sub_iter1 : deletes pixels ...

Input:
img   - list: binary image

Output:
c   - int: 1 if change occured, 0 if change does not occur
'''
def sub_iter1(img):
   print('sub iteration 1')
   to_be_deleted = []
   c = 0                # whether changes are being made or not
   IT = img             # already binary here
   rows , cols = IT.shape

   for i in range(1,rows-1):
       for j in range(1,cols-1):
           nbhd = find_neighbourhood(i, j, IT)

           # check conditions
           a = 2 <= find_nonzero_nbs(nbhd) <= 6
           b = find_transitions(nbhd) == 1
           c_d = find_product(nbhd, 1) == (0, 0)

           conditions = a * b * c_d  # AND GATE with all the conditions

           if IT[i, j] == 1 and conditions:
               to_be_deleted.append((i, j))  # store pixel for deletion
               c = 1

   new_img = delete_p(to_be_deleted, IT)

   return new_img, c
```

This is the second sub iteration

```
def sub_iter2(img):
   print('sub iteration 2')
   to_be_deleted = []
   c = 0                # whether changes are being made or not
   IT = img             # already binary here
   rows , cols = IT.shape

   for i in range(1,rows-1):
       for j in range(1,cols-1):
           nbhd = find_neighbourhood(i, j, IT)

           # check conditions
           a = 2 <= find_nonzero_nbs(nbhd) <= 6
           b = find_transitions(nbhd) == 1
           c_d = find_product(nbhd, 2) == (0, 0)

           conditions = a * b * c_d  # AND GATE with all the conditions

           if IT[i, j] == 1 and conditions:
               to_be_deleted.append((i, j))  # store pixel for deletion
               c = 1

   # delete points
   new_img = delete_p(to_be_deleted, IT)

   return new_img, c
```

This is the algorithm put together

```
def z_s_algo(img_gray):

    img = cv2.threshold(img_gray, 170, 1, cv2.THRESH_BINARY_INV)[1] #[0] = thresh,
[1] = img cv2.threshold(img_gray, 170, 1, cv2.THRESH_BINARY)[1]
    print(img.shape)
    i = 0

    c1 = 1
    c2 = 1

    while c1 or c2:
        print('iteration', i)
        new_img, c1 = sub_iter1(img)
        final_img, c2 = sub_iter2(new_img)
        i+=1

    thinned[thinned == 0] = 255
    thinned[thinned == 1] = 0
    final = cv2.cvtColor(thinned, cv2.COLOR_GRAY2RGB)
    cv2.imwrite('final_grey.png', final)

    return final_img
```

Results

After applying 25 iterations of the Zhang-Suen algorithm, a 416x182 pixel image of the handwritten word, “CP467” is shown below. The algorithm proved to be successful and maintained the objectives of thinning; connectedness, and endpoint retention.

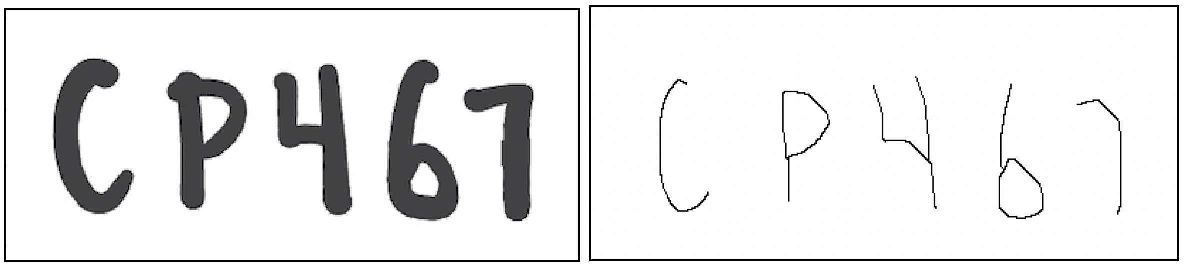

Figure 2.0 - Final Image after 25 Iterations

A visual depiction of the different iteration levels are also shown below.

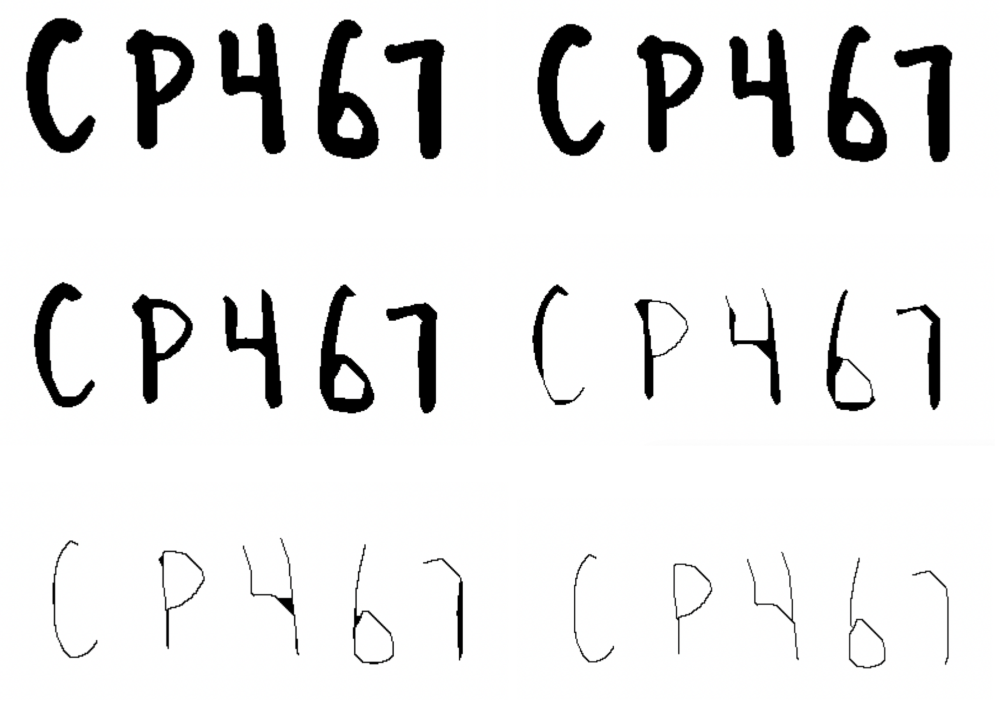

Figure 2.1 - Image after Iteration 1, 2, 5, 10 15 and 25

Issues with Algorithm

Although the thinning method was successful, I observed a few particular issues that can be further investigated to find a solution.

1. **Noise**

   As mentioned in, (Lü & Wang, 1986), if noise is added to the original image, the resulting image not only considers the noise but amplifies it. For instance, in the figure below, the noise is not removed and significantly alters the number “4” in the image.

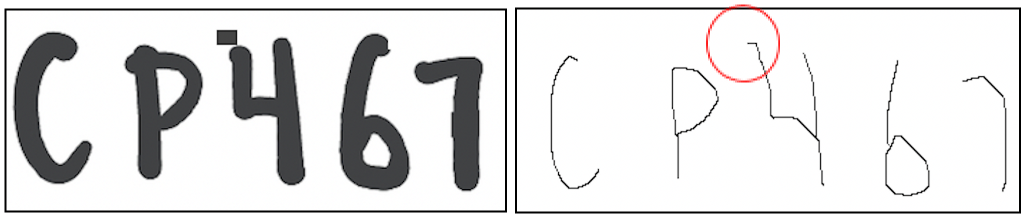

Figure 3.0 - Result of noise in image

2. **Excessive Thinning**

   For some images, the thinning was not satisfactory, for instance, there is excessive thinning when there is a diagonal line consisting of 2 pixels.

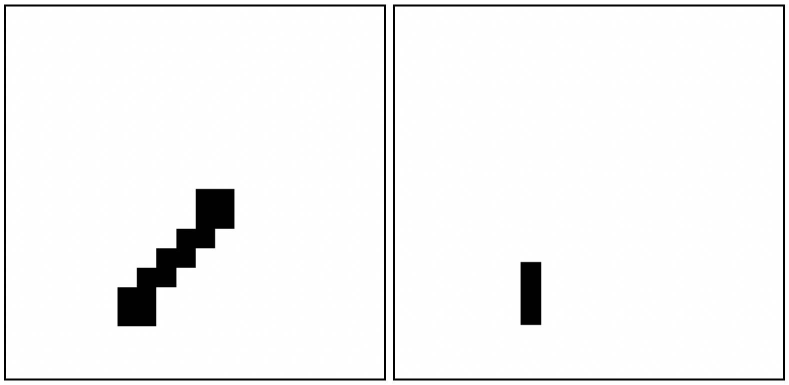

Figure 3.1 - Result of thinning a diagonal line

References

Sawhney, R. *Medial axis transform*. Retrieved December 4, 2022, from http://www.rohansawhney.io/medial-axis-transform.pdf

Lü, H. E., & Wang, P. S. (1986). A comment on “a fast parallel algorithm for thinning digital patterns.” *Communications of the ACM*, *29*(3), 239–242. https://doi.org/10.1145/5666.5670

Zhang, T. Y., & Suen, C. Y. (1984). A fast parallel algorithm for thinning digital patterns. *Communications of the ACM*, *27*(3), 236–239. https://doi.org/10.1145/357994.358023