Project 8 (C++): You are to implement the thinning algorithm – the second method to obtain the skeletons of objects in a given image. The thinning topic is also exhibited in TA’s lecture notes of Medial Axis/Skeleton.

The thinning of an object is like peeling off one layer of object from 4 sides (north, south, west, and east) in iterations, until the object becomes a skeleton. An object pixel can be peel off (turn from 1 to 0) needs to meet certain criteria.

Criteria for thinning from north are:

1) p (i, j) is 1, an object pixel.

2) p (i, j)’s northern neighbor, p (i-1, j) is 0, a background pixel.

3) at least 4 out of 8 p (i, j)’s 3x3 neighbors are 1, object pixels

4) by turning p (i, j) from object pixel to background pixel, i.e., from 1 to 0,

p (i, j)’s neighborhood must remain as one 8-connected component.

For example, if p (i, j)’s 3 by 3 neighbor as show on the left below

If we would to turn p (i, j) from 1 to 0, the result, as show on the right is NO GOOD

1 0 1 1 0 1

0 1 0 🡪 0 0 0

1 0 1 1 0 1

Because we have turned p (i, j)’s neighborhood from 1 8-connected component to 4;

while the example below will be OK.

0 0 1 0 0 1

0 1 1 🡪 0 0 1

1 1 0 1 1 0

You may think that you need to check 27 cases for condition 4; the answer is NO. You only need to check a few configurations of p(i, j)’s neighborhood to determine whether or not to turn p (i, j) from 1 to 0

\*\*\*For your hard copy, you need to draw the configurations you used in your code for checking the conditions 4),

\*\*\* and write the algorithm steps for check3n4Conditions (firstAry, i, j) method, i.e., checking condition 3 and 4.

Criteria for thinning from South is similar to that of thinning from north, except condition 2

2) p (i, j)’s southern neighbor, p (i+1, j) is 0, a background pixel.

Criteria for thinning from West is similar to that of thinning from north, except condition 2

Criteria for thinning from East is similar to that of thinning from north, except condition 2

\*\*\* YOU MUST FOLLOW THE SPEC in your implementation of this project, include all methods given in

\*\*\* the specs and algorithm steps!!

\*\*\* NO POINT will be given if otherwise done!

What you need to do:

- You will have 2 date files: image1, and image2 to test your program.

- Run your program on image1, then run your program on image2

- Include in your hard copies:

- Project cover page

(in addition to your main algorithm steps, include your drawings of 3 by 3 configurations that you used for checking Condition 4 and the algorithm steps for doing so, as described in the above.)

- Project source code

- For image1

- Print outFile1

- Print outFile2

- For image2

- Print outFile1

- Print outFile2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Language: C++

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Points: 10 pts

Due Date: Soft copy and pdf hard copies: 4/5/2020 Sunday before midnight

1 day late: -1 pt 4/6/2020 Monday before midnight

2 days late: -3 pts 4/7/2020 Tuesday before midnight

-10 pts: after 4/7/2020 Tuesday after midnight

\*\*\* Name your pdf file using the same naming convention as given prior

\*\*\* All on-line submission MUST include Soft copy and pdf hard copy

\*\*\* in the same email with correct file names.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

I. Input: inFile (argv [1]): a binary image

II. Outputs:

a) outFile1 (argv [2] ): to store the final thinning result, as an image, with image header.

b) outFile2 (argv [3] ):

- Pretty print the input image with proper caption.

- Pretty print after completing each cycle; a cycle is after thinning all sides.

with proper caption, i.e.,

(“result of thinning : cycle – 1”)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

III. Data structure:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

- A ThinningSkeleton class

- numRows (int)

- numCols (int)

- minVal (int)

- maxVal (int)

- changeflag (int)

- cycleCount (int)

- firstAry (int \*\*) // a 2D array, need to dynamically allocate at run time

of size numRows + 2 by numCols + 2.

- secondAry (int \*\*) // a 2D array, need to dynamically allocate at run time

of size numRows + 2 by numCols + 2.

- methods:

- zeroFrame (…) // framing the extra rows and extra columns with zeros.

- loadImage (inFile, firstAry)

// Read from pixels from inFile onto firstAry begins at firstAry[1][1]

- copyArys (firstAry, secondAry) // always copy from secondAry to firstAry

- doThinning (…) // see algorithm below

- northThinning (…) // See algorithm below

- southThinning (…)// You should know how to do this

- westThinning (…) // You should know how to do this

- eastThinning (…) // You should know how to do this

- (bool) check3n4Conditions (firstAry, i, j)

// see the checking conditions 3 and 4 as describe in the above.

// You need to write your algorithm steps and include it in your hard copy

- prettyPrint(…)

// if Ary(i, j) > 0

outFile2 🡨output Ary(i, j) plus one blank space

else

outFile2 🡨output two blank spaces.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

IV. main (...)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

step 0: inFile 🡨 open from argv

outFile1, outFile2🡨 open from argv

step 1: numRows, numCols, minVal, maxVal 🡨 read from inFile

outFile1 🡨 output numRows, numCols, minVal, maxVal to outFile1

dynamically allocate firstAry of size numRows + 2 by numCols + 2.

dynamically allocate secondAry of size numRows + 2 by numCols + 2.

step 2: zeroFrame(firstAry)

zeroFrame(secondAry)

step 3: loadImage (inFile, firstAry)

step 4: prettyPrint (firstAry, outFile2) // This print is before thinning

step 5: changeFlag 🡨 0

step 6: doThinning (firstAry, secondAry, changeFlag)

Step 7: prettyPrint (firstAry, outFile2)

Step 8: repeat step 5 to step 7 while changeFlag > 0

step 9: outFile1 🡨 output firstAry from [1][1] \*without\* extra rows and cols

step 10: close all files

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

V. doThinning (firstAry, secondary, changeFlag)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

step 1: northThinning (firstAry, secondary, changeFlag) // Algorithm is given below.

copyArys (firstAry, secondAry) // always copy from secondAry to firstAry

step 2: southThinning (firstAry, secondAry, changeFlag) // on your own

copyArys (firstAry, secondAry) // always copy from secondAry to firstAry

step 3: westThinning (firstAry, secondAry, changeFlag) // on your own

copyArys (firstAry, secondAry) // always copy from secondAry to firstAry

step 4: EastThinning (firstAry, secondAry, changeFlag) // on your own

copyArys (firstAry, secondAry) // always copy from secondAry to firstAry

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

VI. NorthThinning (firstAry, secondAry, changeFlag)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// The algorithm steps may contain bug, it is for you to debug

step 1: scan the firstAry L to R & T to B, begins at (1,1), and ends at (?, ?)

// use index i for row and index j for column

step 2: firstAry(i,j) 🡨 next pixel

step 3: secondAry(i,j) 🡨 firstAry(i, j)

step 4: if firstAry(i, j) > 0 && firstAry(i-1, j) <= 0 // northern neighbor is a background pixel

if ( check3n4Conditions (firstAry, i, j) == true )

secondAry(i,j) 🡨 0

changeFlag ++

Step 5: repeat step 1 to step 4 until all pixels are processed.