Project 8 (C++): Document text-line detection using Projection Profiles of the given document zone.

As taught in class, one major task in the document image analysis is to decompose a given document into a hierarchical tree structures where the root is the whole document (could be one page or multiple pages); the next level in the hierarchy under a page are one or more column blocks, each column block could consists of text zones and other none-text zones (such as figures, graphic, table, math equations, ...); the next level under text-zones are paragraphs; below paragraph are text-lines; below text-lines are text-words; below text-words are characters. An Optical Character Recognition (OCR) system begins it document recognition by image decomposition to form a document hierarchy, then from bottom up of the document hierarchical tree, OCR first performs character recognition, then, up to form words, up to form text-lines, and so for. A highly effective technique for document image decomposition is using the projection profiles of a given document to construct the document hierarchy top-down.

As taught in class, the HPP and VPP can also be used to determine the reading direction of a given document, by analyzing the patterns of HPP and VPP and to determine text-zones vs non-text zones.

In this project, the input image contains a single "zone" from a document. If the zone is a none-text zone, your program will say so. If the zone is a text-zone, your program will do the followings: 1) compute HPP and VPP of the zone; 2) determine the zone bounding boxes from HPP and VPP; 3) overlay and output the zone bounding boxes onto the input image; 4) determine the reading direction of the document; 5) determine and overlay the bounding boxes of text-lines within the zone.

To accomplish the three tasks given in the above, your program will perform the followings:
1) Computes the two project profiles of the input zone image.
2) Binarizes the two project profiles via thresholding, using threshold value at 3, 4, or 5, to eliminate the background noises.
3) Determines the zone bounding box based on the two binarized project profiles.
4) Performs 1D morphological closing on the two binarized project profiles to eliminate foreground noises.
5) Counts the number of runs of 1's on the two morphological processed projection profiles.
6) Determine the reading direction based on the number of runs within the two morphological processed projection profile.
7) Extract text-line bounding boxes within the zone using the two morphological processed projection profiles.
8) Overlay the zone bounding box and the extracted text-line bounding boxes on the document zone image.

==========

*** You will be given 3 data files: zone1, zone2 and zone3. zone1 contains a horizontal text zone; zone2 contains a vertical text zone and zone3 contains a non-text zone.

What to do as follows:
1) Implement your program based on the specs given below.
2) Run and debug your program on zone1 until your program says the text reading direction is horizontal and produces correct bounding boxes.
2) Run and debug your program on zone2 until your program says the text reading direction is vertical and produces correct bounding boxes.
4) Run and debug your program on zone3 until your program says it contains a non-text zone.

Include in your hard copies:
- cover page
- source code
- Pretty Print zone1
- outFile1 for zone1
- outFile 2 for zone1
- Pretty Print zone2
- outFile1 for zone2
- outFile 2 for zone2
- Pretty Print zone3
- outFile1 for zone3
- outFile 2 for zone3

****

Language: C++

****

Project points:12 pts

Due Date: Soft copy (*.zip) and hard copies (*.pdf):

+1 (13/12 pts): early submission, 5/8/2023, Monday before midnight

-0 (12/12 pts): on time, 5/12/2023, Friday before midnight. NO LATE submission!

*** Name your soft copy and hard copy files using the naming convention as given in the project submission requirement.
*** All on-line submission MUST include soft copy (*.zip) and hard copy (*.pdf) in **the same email attachments** with correct email subject as stated in the email requirement; otherwise, your submission will be rejected.
** You will be given two data files, run your program on each file. Print both results in your hard copies.

****

I. Inputs: a) inFile1 (argv [1]): A binary image.

b) a threshold value (argv [2]): For noise cleaning. Try use ThrValue = 3, 4, 5

c) structElemFile (argv [3]) // You may hardcode in your program for the structuring element as 1 **1** 1
// or make a structuring element file as below, and use argv [3]

```
1  3  1  1 // 1 rows, 3 columns, min is 1, max is 1:
0  1        // origin is at row index 0 and column index 1.
1  1  1
```

****

II. outFiles:

a) outFile1 (argv [3]): as directed in the specs below.

b) outFile2 (argv [4]): as directed in the specs below.

****

III. Data structure:

****

- a boxNode class
  - (int) boxType // 1 for zone; 2 for text-line.
  - (int) minR
  - (int) minC
  - (int) maxR
  - (int) maxC
  - (boxNode*) next // points to boxNode in the same level.

  method:
  - constructor (…)
- a docImage class:
  - (int) numRows
  - (int) numCols
  - (int) minVal
  - (int) maxVal
  - (int) numStructRows // if not using file, you may hardcode it.
  - (int) numStructCols // if not using file, you may hardcode it.
  - (int) structMin // if not using file, you may hardcode it.
  - (int) structMax // if not using file, you may hardcode it.
  - (int) rowOrigin // if not using file, you may hardcode it.
  - (int) colOrigin // if not using file, you may hardcode it.
  - (int **) imgAry // a 2D array, dynamically allocate, size of numRows + 2 by numCols + 2
    // You need to zero-framed the imgAry, unlike java, C++ does not initialize.
  - (int) structElem [3] // 1**1**1, either hardcoded or read from argv [3]
  - (int*) HPP // a 1-D array, size of numRows +2, to store the horizontal projection profile, dynamically allocate,
    // initialize to 0.
  - (int*) VPP // a 1-D array, size of numCols +2, to store the vertical projection profile, dynamically allocate,
    // initialize to 0.
  - (int*) binHPP // a 1-D array of the binarized HPP, dynamically allocate at run time, size as HPP.
  - (int*) binVPP // a 1-D array to store the binarized VPP, dynamically allocate at run time, size as VPP.

- (int*) morphHPP // a 1-D array to store the result of 1D morphological closing of binHPP, dynamically
// allocate, size as HPP
- (int*) morphVPP // a 1-D array to store the result of 1D morphological closing of binVPP, dynamically
// allocate, size as VPP
- (boxNode*) listHead // The head of boxNode linked list points to a dummy node. The list does not need to be sorted
// Insertion will take place after dummy.
- (int) thrVal // the threshold value provided in argv [2]
- (int) runsHPP // The number of runs in morphHPP.
- (int) runsVPP // The number of runs in morphVPP.
- (int) readingDirection // 1 for horizontal, 2 for vertical.

- methods:
- constructor (…) // establishes, allocates and initializes all members of docImage class
- loadImage (inFile, imgAry) // Load the inFile inside of framed of imgAry, begins at [1][1]
- computePP (imgAry) // computes the horizontal and vertical projection profiles of object pixels of imgAry
// and stores in HPP and VPP. On your own.

- binaryThreshold (PP, thrVal, binPP) // Performs a binary threshold on the given projection profile, PP

- printPP (PP, outFile) // prints the given projection profile to outFile (can be outFile1 or outFile2).
- (boxNode *) computeZoneBox (…) // see algorithm below.

- morphClosing (PP, structElem, morphPP) // computes 1D morphological closing, using 1**1**1 as the structing element
//on the given PP and store in morphPP. Reuse codes from your morphology project.

- listInsert (listHead, Bnode) // Inserts Bnode after the dummy node. No need to use findSpot ().
// Reuse codes from your graph coloring project.

- (int) computePPruns (…) // see algorithm below.
// computes the number of run in morphPP, labelling each run, in sequence: 1, 2, 3, …
// overwriting morphPP and returns the number of runs.

- (int) computeDirection (runsHPP, runsVPP) // see algorithm below; determines the reading direction.
- computeHorizontalTextBox (…)// see algorithm below.
// compute the bounding box of text-lines for horizonal text, in sequence;
// and insert each text-line box at the front of the linked list, after dummy.

- computeVerticalTextBox (…)// see algorithm below.
// compute the bounding box of text-lines for vertical text, in sequence;
// and insert each text-line box at the front of the linked list, after dummy.

- imgReformat (…) Reuse codes from your previous projects.

- overlayBox (listHead, imgAry) // overlay each bounding box of boxNode in the linked list onto imgAry.
// Reuse codes from your connected component project, use 9 for boundary value.

- printBox (listHead, outFile) // print boxNode in the linked list to outFile (can be outFile1 or outFile2)
//in the format below:
box type // 1 or 2. 1 for zone, 2 for text-line
minRow minCol maxRow maxCol // bounding box
box type
minRow minCol maxRow maxCol // bounding box
:

For example:

==========

The following are bounding box for the input zone:

2
3 1 45 46
2
3 1 4 46
1
11 1 18 46

****
IV. main (…)
****

Step 0: open all files from argv[]
thrVal ←argv[2]
numRows, numCols, minVal, maxVal ← inFile
numStructRows, numStructCols, StructMin, StructMax, rowOrigin, colOrigin ← structElemFile or hard coded
use constructor to establish, allocate, and initialize all members of docImage class; unlike Java, C++ does NOT do any initialization; therefore, make sure to initialize as indicate in the date structure in the above.

Step 1: loadImage (inFile, imgAry)
outFile1 ← " Below is the input image"
imgReformat (imgAry, outFile1)

Step 2: computePP (imgAry)
outFile2 ← "Below is HPP"
printPP (HPP, outFile2)
outFile2 ← "Below is VPP"
printPP (VPP, outFile2)

Step 3: binaryThreshold (HPP, thrVal, binHPP)
binaryThreshold (VPP, thrVal, binVPP)
outFile2 ← "Below is binHPP"
printPP (binHPP, outFile2)
outFile2 ← "Below is binVPP"
printPP (binVPP, outFile2)

Step 4: listHead ← get a boxNode, as the dummy node for listHead to point to.
(boxNode*) zBox ← computeZoneBox (binHPP, binVPP)
listInsert (listHead, zBox) // insert zBox to the front of linked list, after dummy
outFile2 ← "Below is the linked list after insert input zone box"
printBox (listHead, outFile2)

Step 5: morphClosing (binHPP, structElem, morphHPP)
morphClosing (binVPP, structElem, morphVPP)
outFile2 ← "Below is morphHPP after performing morphClosing on HPP"
outFile2 ← printPP (morphHPP)
outFile2 ← "Below is morphVPP after performing morphClosing on VPP"
printPP (morphVPP)

Step 6: runsHPP ← computePPruns (morphHPP, numRows)
runsVPP ← computePPruns (morphVPP, numCols)
outFile2 ← The number of runs in morphHPP-runsHPP is " // fill in value.
outFile2 ← The number of runs in morphVPP – runsVPP is " // fill in value.

Step 7: readingDirection ← computeDirection (runsHPP, runsVPP)
outFile2 ← "readingDirection is" // fill in value.

Step 8: if readingDirection == 1
computeHorizontalTextBox (zoneBox, morphHPP, numRows)
else if readingDirection == 2
computeVerticalTextBox (zoneBox, morphVPP, numCols)

Step 9: overlayBox (listHead, imgAry)

Step 10: outFile1 ← "Below is the input image overlay with bounding boxes"
imgReformat (imgAry)

Step 11: outFile1 ← "Output the boxNode in the list"
printBox (listHead, outFile1)

Step 12: close all files.

```
******************************
V. (boxNode*) computeZoneBox (binHPP, VPPBin)
******************************
Step 0:  minR ← 1
         minC ← 1
         maxR ←numRows
         maxC ←numCols
Step 1: if binHPP [minR] == 0   // processing from the beginning of binHPP until reach a non-zero
           minR++
Step 2: repeat Step 1 while binHPP [minR] == 0 && minR <= numRows
Step 3: if binHPP [maxR] == 0 // Processing from the ending of binHPP until reach a non-zero
           maxR--
Step 4: repeat Step 3 while binHPP [maxR] == 0 && maxR >= 1
Step 5: if binVPP [minC] == 0
           minC++
Step 6: repeat Step 5 while binVPP [minC] == 0 && minC <= numCols
Step 7: if binVPP [maxC] == 0
           maxC--
Step 8: repeat Step 7 while  binVPP [maxC] == 0 && maxC >= 1
Step 9:  B ← get a boxNode (1, minR, minC, maxR, maxC, null) // B points to the created boxNode
Step 10: return B

******************************
VI. (int) computePPruns (PP, lastIndex)
******************************
Step 0: numRuns ←0
         index ← 1
Step 1: if PP[index] == 0 // skipping 0's
         index++
Step 2: repeat Step 1 while PP[index] == 0 && index <= lastIndex
Step 3: if PP[index] > 0 // counting consecutive 1's
Step 4: index++
Step 5: repeat Step 3 to Step 4 while PP[index] > 0 && index <= lastIndex
Step 6: numRuns ++
Step 7: repeat Step 1 to Step 7 while index <= lastIndex
Step 8: return numRuns

******************************
VII. (int) computeDirection (runsHPP, runsVPP)
******************************
Step 0: factor ← 2  //
        direction ← 0
Step 1: if runsHPP <= 2 && runsVPP <= 2
                  outFile1 ← Output “the zone may be a non-text zone”
         else if runsHPP >= factor * runsVPP
                  outFile1 ← Output " the document reading direction is horizontal!"
                  direction ← 1
         else if runsVPP >= factor * runsHPP
                  outFile1 ← Output " the document reading direction is vertical!"
                  direction ← 2
         else outFile1 ← Output “the zone may be a non-text zone”

Step 2: return direction
```

```
*****************************
VIII. computeHorizontalTextBox (zBox, PP, lastIndex)
*****************************
Step 0:  minR ← zoneBox's minR
         minC ← zoneBox's minC
         maxR ← minR // Start at the beginning.
         maxC ←zoneBox's maxC
Step 1: if PP[maxR] == 0 // skip leading 0's
                  maxR ++
Step 2: repeat Step 1 while PP[maxR] == 0 && maxR <= lastIndex
                  minR ← maxR  // update minR
Step 3: if PP[maxR] > 0
                  maxR ++
Step 4: repeat Step 3 while PP[maxR] > 0 && maxR <= lastIndex
Step 5: B ← get a boxNode (2, minR, minC, maxR, maxC, null)
         listInsert (listHead, B)
Step 6: minR ← maxR
Step 7: if PP[minR] == 0 // skip 0's in mid-stream
                  minR++
Step 8: repeat Step 7 while PP[minR] == 0 && minR <= lastIndex
Step 9: maxR ← minR
Step 10: repeat Step 3 to Step 7 while index <= lastIndex

 *****************************
VIII. computeVerticalTextBox (zBox, PP, lastIndex)
*****************************
Step 0:  minR ← zBox's minR
         minC ← zBox's minC
         maxR ← zBox's maxR
         maxC ← minC  //Start at the beginning
Step 1: if PP[maxC] == 0 // skip leading 0's
                  maxC ++
Step 2: repeat Step 1 while PP[maxC] == 0 && maxC <= lastIndex
                  minC ← maxC  // update minC
Step 3: if PP[maxC] > 0
                  maxC ++
Step 4: repeat Step 3 while PP[maxC] > 0 && maxC <= lastIndex
Step 5: B ← get a box (2, minR, minC, maxR, maxC, null)
         listInsert (listHead, B)
Step 6: minC ← maxC
Step 7: if PP[minC] == 0 // skip 0's in mid-stream
                  minC++
Step 8: repeat Step 7 while PP[minC] == 0 && minC <= lastIndex
Step 9: maxC ← minC
Step 10: repeat Step 3 to Step 7 while index <= lastIndex
```