**William Benton**

**Lab 3: Optical Character Recognition with Thinning**

**10/02/2019**

**Background**

The purpose of this lab was to scan an image of text and locate all of a certain letter throughout the entirety of the text. This was done by using an image of the letter to find in that font and using template matching on the image of the text. Template matching will look at each pixel and compare the surrounding pixels to the template. In the case of this lab, the template is the letter ‘e’. Once a letter was found, a copy of a 9x15 area around the pixel in the original input image was saved off. That image was thinned and tested for end points and branch points. An ‘e’ has one branch point and one end point. If the saved off image had 1 end point and 1 break point, then it was marked as an e.

**Implementation**

Since this lab builds on top of the previous lab and the previous lab was successful, I simply started by reading in the MSF image that was created last time. I also read in the same ground truth table. From here, the MSF was put through threshold values ranging from 50-255. I chose 50 randomly because previously there was no differences after a certain low enough threshold. After thresholding, a 9x15 area was searched at each point found in the ground truth table. If there was a pixel that was “on” or equal to 255, that meant that an ‘e’ was found. The next steps were modified for this lab. If an image was read in, a copy of the 9x15 area centered at the coordinates found in the ground truth table was taken from the original input image. An example of this can be seen in Figure 1. This image was taken and put through a thinning algorithm. An example of this can be seen in Figure 2. Once it was thinned, the image was tested to see how many end points and how many break points were in the image. If there was exactly one of each, it was marked as an ‘e’. From here, the results were compared to the ground truth table to determine how well the program performed. These results can be seen and will be A picture containing text, crossword puzzle

Description automatically generatedA picture containing bathroom, next

Description automatically generateddiscussed in the results section below.

Figure : Thinned Image

Figure : Copied Image

**Results**

The ROC Curve can be seen in Figure 3. This curve is very different from the previous ROC curve. This curve never reaches a true positive rate of 1, however it has a very low false positive rate. So, while it missed some letters, it significantly reduced the number of false positives that were encountered.

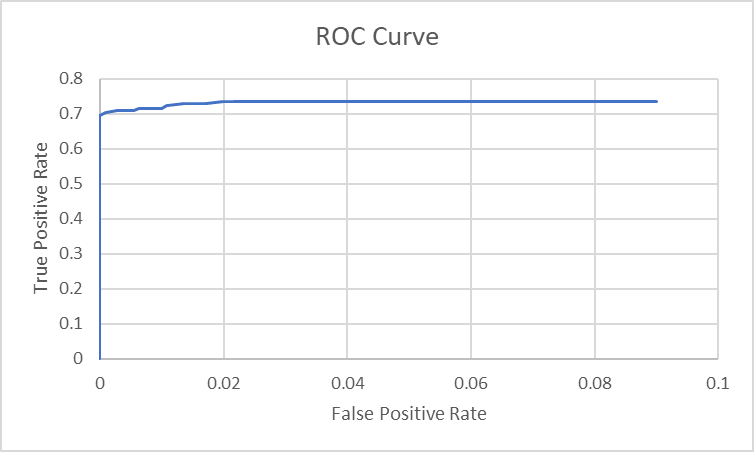


Figure 3: ROC Curve

After calculating the distance for every threshold value, it was determined that the best threshold value was 199 with 111 true positives and 22 false positives. The distance from the corner at [0,1] on the curve for this threshold value was 0.26563975. Figure 4 shows the starting image and figure 5 shows the result of the MSF filter after normalization.

**A screenshot of a cell phone

Description automatically generatedA close up of a newspaper

Description automatically generated**

Figure 5: Normalized MSF Image

Figure 4: Input Image

**Code**

**letters.h**

struct GT\_s{

int Column;

int Row;

char Letter;

};

void find\_letter(unsigned char \*MSF, int ROWS, int COLS, struct GT\_s \*GT, int size, unsigned char \*input\_image);

void thinning(unsigned char \*image);

void create\_copy(unsigned char \*input\_image, unsigned char \*temp\_letter, int COLS, int Row, int Column);

void threshold(unsigned char \*image, unsigned char \*output, int ROWS, int COLS, int thresh\_val);

**letters.c**

/\* William Benton

\* Lab 3: This program builds on the previous lab and will reduce false

\* positives by using thinning and branchpoint and endpoint detection

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#include "letters.h"

int main()

{

FILE \*fpt;

unsigned char \*input\_image;

char header[320];

int ROWS, COLS, BYTES;

unsigned char \*MSF;

struct GT\_s \*GT;

int lettercount = 0;

// Open input image

fpt = fopen("parenthood.ppm", "rb");

if (fpt == NULL)

{

printf("Error opening input image\n");

exit(0);

}

fscanf(fpt, "%s %d %d %d", header, &COLS, &ROWS, &BYTES);

if (strcmp(header, "P5") != 0 || BYTES != 255)

{

printf("Error: Image not a greyscale 8-bit PPM image\n");

exit(0);

}

input\_image = (unsigned char \*)calloc(ROWS\*COLS, sizeof(unsigned char));

MSF = (unsigned char \*)calloc(ROWS\*COLS, sizeof(unsigned char));

header[0] = fgetc(fpt);

fread(input\_image, 1, COLS\*ROWS, fpt);

fclose(fpt);

fpt = fopen("MSF.ppm", "rb");

if (fpt == NULL)

{

printf("Error opening MSF image\n");

exit(0);

}

fscanf(fpt, "%s %d %d %d", header, &COLS, &ROWS, &BYTES);

header[0] = fgetc(fpt);

fread(MSF, 1, COLS\*ROWS, fpt);

fclose(fpt);

// Read in Ground Truth

fpt = fopen("parenthood\_gt.txt", "rb");

if (fpt == NULL)

{

printf("Error opening ground truth table\n");

exit(0);

}

GT = (struct GT\_s \*)calloc(1300, sizeof(struct GT\_s));

while(!feof(fpt))

{

fscanf(fpt, "%c %d %d\n", &GT[lettercount].Letter, &GT[lettercount].Column, &GT[lettercount].Row);

lettercount++;

}

find\_letter(MSF, ROWS, COLS, GT, lettercount, input\_image);

return 1;

}

void threshold(unsigned char \*image, unsigned char \*output, int ROWS, int COLS, int thresh\_val)

{

int i;

for (i = 0; i < ROWS\*COLS; i++)

{

if (image[i] > thresh\_val)

{

output[i] = 255;

}

else

{

output[i] = 0;

}

}

return;

}

void create\_copy(unsigned char \*input\_image, unsigned char \*temp\_letter, int COLS, int Row, int Column)

{

int i, j, k, l;

k = 0;

l = 0;

for (i = Row - 7; i <= Row + 7; i++)

{

k = 0;

for (j = Column - 4; j <= Column + 4; j++)

{

temp\_letter[l\*9+k] = input\_image[i\*COLS+j];

k++;

}

l++;

}

return;

}

void thinning(unsigned char \*image)

{

int row, col;

int i;

// Edge to non edge

int E2NE;

int neighbors;

bool erase\_flag;

// Flag to keep track of edge to non edge

bool prev;

unsigned char \*copy;

copy = (unsigned char \*)calloc(9\*15, sizeof(unsigned char));

// Threshold Image first

for (i = 0; i < 9\*15; i++)

{

if (image[i] < 128)

{

image[i] = 255;

}

else

{

image[i] = 0;

}

}

erase\_flag = true;

while (erase\_flag)

{

for (i = 0; i < 9\*15; i++)

{

copy[i] = 0;

}

erase\_flag = false;

for (row = 1; row < 15; row++)

{

for (col = 1; col < 9; col++)

{

i = row\*9+col;

// Skip if pixel is not "on"

if (!(image[i] == 0))

{

// Check clockwise for neighbors and E2NE transitions

E2NE = 0;

neighbors = 0;

// North

if (image[i-9] == 255)

{

neighbors++;

prev = true;

}

else

{

prev = false;

}

// North East

if (image[i-9+1] == 255)

{

neighbors++;

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// East

if (image[i+1] == 255)

{

neighbors++;

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// South East

if (image[i+1+9] == 255)

{

neighbors++;

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// South

if (image[i+9] == 255)

{

neighbors++;

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// South West

if (image[i+9-1] == 255)

{

neighbors++;

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// West

if (image[i-1] == 255)

{

neighbors++;

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// North West

if (image[i-1-9] == 255)

{

neighbors++;

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// Check the North West to North Transition for E2NE

if ((image[i-9] == 0) && prev)

{

E2NE++;

}

// Check to see if pixel should be erased

if (E2NE == 1)

{

if (neighbors >= 3 && neighbors <= 7)

{

if ((image[i-9] == 0) || (image[i+1] == 0) || ((image[i+9] == 0) && (image[i-1] == 0)))

{

copy[i] = 255;

erase\_flag = true;

}

}

}

}

}

}

for (i = 0; i < 9\*15; i++)

{

if (copy[i] == 255)

{

image[i] = 0;

}

}

}

return;

}

bool end\_and\_branch\_detect(unsigned char \*image)

{

int i, row, col;

int E2NE;

int endpoint, branchpoint;

bool e\_found;

bool prev;

endpoint = 0;

branchpoint = 0;

for (row = 1; row < 15; row++)

{

for (col = 1; col < 9; col++)

{

i = row\*9+col;

// Skip if pixel is not "on"

if (!(image[i] == 0))

{

// Check clockwise for neighbors and E2NE transitions

E2NE = 0;

// North

if (image[i-9] == 255)

{

prev = true;

}

else

{

prev = false;

}

// North East

if (image[i-9+1] == 255)

{

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// East

if (image[i+1] == 255)

{

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// South East

if (image[i+1+9] == 255)

{

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// South

if (image[i+9] == 255)

{

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// South West

if (image[i+9-1] == 255)

{

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// West

if (image[i-1] == 255)

{

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// North West

if (image[i-1-9] == 255)

{

prev = true;

}

else

{

if (prev)

{

E2NE++;

}

prev = false;

}

// Check the North West to North Transition for E2NE

if ((image[i-9] == 0) && prev)

{

E2NE++;

}

if (E2NE == 1)

{

endpoint++;

}

else if (E2NE > 2)

{

branchpoint++;

}

}

}

}

if (endpoint == 1 && branchpoint == 1)

{

e\_found = true;

}

else

{

e\_found = false;

}

return e\_found;

}

void find\_letter(unsigned char \*MSF, int ROWS, int COLS, struct GT\_s \*GT, int Size, unsigned char \*input\_image)

{

int i, j, k;

unsigned char \*temp;

int thresh\_val;

FILE \*fpt1;

FILE \*fpt2;

int tp, fp, tn, fn;

bool found;

unsigned char \*temp\_letter;

bool e\_found;

temp = (unsigned char \*)calloc(ROWS\*COLS, sizeof(unsigned char));

temp\_letter = (unsigned char \*)calloc(15\*9, sizeof(unsigned char));

for (thresh\_val = 50; thresh\_val < 255; thresh\_val++)

{

threshold(MSF, temp, ROWS, COLS, thresh\_val);

// Prepare excel file

fpt2 = fopen("ROC.csv", "w");

fprintf(fpt2, ",TP,FP,TN,FN\n");

fpt2 = fopen("ROC.csv", "a");

tp = 0;

fp = 0;

tn = 0;

fn = 0;

for (i = 0; i < Size; i++)

{

found = false;

e\_found = false;

for (j = GT[i].Row - 7; j <= GT[i].Row + 7 && found == false; j++)

{

for (k = GT[i].Column - 4; k <= GT[i].Column + 4 && found == false; k++)

{

if (temp[j\*COLS+k] == 255)

{

create\_copy(input\_image, temp\_letter, COLS, GT[i].Row, GT[i].Column);

thinning(temp\_letter);

e\_found = end\_and\_branch\_detect(temp\_letter);

found = true;

}

}

}

if (e\_found == true && GT[i].Letter == 'e')

{

tp++;

}

else if (e\_found == true && GT[i].Letter != 'e')

{

fp++;

}

else if (e\_found != true && GT[i].Letter == 'e')

{

fn++;

}

else if (e\_found != true && GT[i].Letter != 'e')

{

tn++;

}

}

fprintf(fpt2, "%d, %d, %d, %d, %d\n", thresh\_val, tp, fp, tn, fn);

}

// Added after to include examples

create\_copy(input\_image, temp\_letter, COLS, 25, 55);

fpt1 = fopen("CopiedImage.ppm", "w");

fprintf(fpt1, "P5 %d %d 255\n", 9, 15);

fwrite(temp\_letter, 9\*15, 1, fpt1);

fclose(fpt1);

thinning(temp\_letter);

fpt1 = fopen("Thinned.ppm", "w");

fprintf(fpt1, "P5 %d %d 255\n", 9, 15);

fwrite(temp\_letter, 9\*15, 1, fpt1);

fclose(fpt1);

return;

}

**References**

ROC Calculations:

[1] <https://www.medcalc.org/manual/roc-curves.php>

[2] <https://ncss-wpengine.netdna-ssl.com/wp-content/themes/ncss/pdf/Procedures/NCSS/One_ROC_Curve_and_Cutoff_Analysis.pdf>