Name: Rodrigo Ignacio Rojas Garcia
Course: ECE 4310
Lab #3

Letters

In this project the student was to implement thinning, branch point, and endpoint detection in order to recognize letters in an image of a text. The student was provided with an input image “parenthood.ppm”, template image of desired character to be found “parenthood_e_template.ppm”, ground truth file “ground_truth.txt”, and was required to use Match-Spatial Filter (MSF) image from laboratory 2 “msf_e.ppm”. All of these images were used to determine if the desired letter was recognized from the MSF.

The third laboratory was divided into multiple steps:

1. Read in input image, template image, MSF image, and ground truth text file.
2. Threshold input image at a value of 128 to create a binary image.
3. Thin the thresholded image down to single-pixel wide components.
4. Determine end point and branch points and store it in a binary image.
5. Loop through the following steps for a range of T (Threshold):
    1. Threshold MSF image into a binary image based on T.
    2. Loop through ground truth letter locations.
    3. Determine if the letter is an “e” or not “e” based on T and if and only if the found “e” has exactly one endpoint and one branch point, if not letter is considered as not an “e”. If letter considered an “e”, add to value TP (“detected and letter is “e”), if not add to value FP (“detected” but the letter is not “e”).

All results of each of these steps were recorded. All images were saved as gray-scale ppm images. Also, the total output of the total FP, TP, FN, FP, TPR, FPR, and PPV were recorded with their corresponding threshold and documented into a CSV (Comma Separated Variable) file.

STEP 1:

Read in “parenthood.ppm”, “parenthood_e_template.ppm”, and “msf_e.ppm” PPM files

```c
/* READS IN IMAGE */
unsigned char *read_in_image(int rows, int cols, FILE *image_file)
{
    // Variable Declaration Section
    unsigned char *image;

    image = (unsigned char *)calloc(rows * cols, sizeof(unsigned char));

    fread(image, sizeof(unsigned char), rows * cols, image_file);

    fclose(image_file);

    return image;
}
```

Read in "ground_truth.txt" text file

```
// Read in Ground Truth text file
file = fopen(file_name, "r");
if (file == NULL)
{
    printf("Error, could not read Ground Truth text file\n");
    exit(1);
}
```

STEP 2

Threshold "parenthood.ppm" PPM file

```
/* THRESHOLD ORIGINAL IMAGE AT VALUE OF 128 */
unsigned char *original_image_threshold(unsigned char *original_image, int original_image_rows, int original_image_cols)
{
    // Variable Declaration Section
    int c1;
    int threshold = 128;
    unsigned char *output_threshold_image;

    // Allocate memory for temporary image
    output_threshold_image = (unsigned char *)calloc(original_image_rows * original_image_cols, sizeof(unsigned char));

    for (c1 = 0; c1 < (original_image_rows * original_image_cols); c1++)
    {
        if (original_image[c1] <= threshold)
        {
            output_threshold_image[c1] = 255;
        }
        else
        {
            output_threshold_image[c1] = 0;
        }
    }

    return output_threshold_image;
}
```

STEP 3

Thin thresholded image down to pixel-wide components

- *Function "thinning()" calls function "get_transitions()" which determine if the pixel needs to be erased. The "get_transitions()" function also calculates both end points and branch points on the thresholded image, but in this case they are not used.*

```
/* THINNING OF ORIGINAL THRESHOLD IMAGE */
unsigned char *thinning(unsigned char *image, int image_rows, int image_cols)
{
    // Variable Declaration Section
    int c1, row, col;
    int is_pixel_marked = 0;
    int index = 0;
    int run_again  = 1;
    int end_points, branch_points;
    unsigned char *thinned_image;
    unsigned char *temp_image;
    int marked_pixels = 0;
    int ran_times = 0;
    end_points = branch_points = 0;

    // Allocate memory for thined image
    thinned_image = (unsigned char *)calloc(image_rows * image_cols, sizeof(unsigned char));
    temp_image = (unsigned char *)calloc(image_rows * image_cols, sizeof(unsigned char));

    // Copy original image to allocated thined image
    for (c1 = 0; c1 < (image_rows * image_cols); c1++)
    {
        thinned_image[c1] = image[c1];
        temp_image[c1] = image[c1];
    }

    // Thinning Algorithm
    while(run_again == 1)
    {
        run_again = 0;
        is_pixel_marked = 0;
        marked_pixels = 0;
        ran_times++;
```

```
        for (row = 1; row < (image_rows - 1); row++)
        {
            for (col = 1; col < (image_cols - 1); col++)
            {
                index = (row * image_cols) + col;
                if (thinned_image[index] == 255)
                {
                    is_pixel_marked = 0;
                    get_transitions(thinned_image, image_rows, image_cols, row,
                    col, &is_pixel_marked, &end_points, &branch_points);
                    if (is_pixel_marked == 1)
                    {
                        index = (row * image_cols) + col;
                        temp_image[index] = 0;
                        run_again = 1;
                        marked_pixels += 1;
                    }
                }
            }
        }

        printf("Pixels Marked on round %d: %d pixels\n", ran_times, marked_pixels);

        for (c1 = 0; c1 < (image_rows * image_cols); c1++)
        {
            thinned_image[c1] = temp_image[c1];
        }
    }
    return thinned_image;
}
```

```
/* DETERMINES IF CURRENT PIXEL OF IMAGE NEEDS TO BE ERASED/KEPT FROM THINNING IMAGE */
void get_transitions(unsigned char *image, int image_rows, int image_cols, int row, int col, int *mark_pixel, int *end_points, int *branch_points)
{
    // Variable Declaration Section
    int A, B, C, D;
    int c1;
    int index = 0;
    int index2 = 0;
    int edge_to_nonedge = 0;
    int edge_neighbors = 0;
    int current_pixel = 0;
    int next_pixel = 0;
    A = B = C = D = 0;

    // Up Row clockwise
    for (c1 = (col - 1); c1 <= col; c1++)
    {
        index = ((row - 1) * image_cols) + c1;
        current_pixel = image[index];
        index2 = ((row - 1) * image_cols) + (c1 + 1);
        next_pixel = image[index2];

        if ((current_pixel == EDGE) && (next_pixel == NOT_EDGE))
        {
            edge_to_nonedge++;
        }
        if (current_pixel == EDGE)
        {
            edge_neighbors++;
        }
    }

    // Right-Down Row Clock-wise
    for (c1 = (row - 1); c1 <= row; c1++)
    {
        index = (c1 * image_cols) + (col + 1);
        current_pixel = image[index];
        index2 = ((c1 + 1) * image_cols) + (col + 1);
        next_pixel = image[index2];
```

```
        if ((current_pixel == EDGE) && (next_pixel == NOT_EDGE))
        {
            edge_to_nonedge++;
        }
        if (current_pixel == EDGE)
        {
            edge_neighbors++;
        }
    }

    // Bottom-Left Row Clock-Wise
    for (c1 = (col + 1); c1 > (col - 1); c1--)
    {
        index = ((row + 1) * image_cols) + c1;
        current_pixel = image[index];
        index2 = ((row + 1) * image_cols) + (c1 - 1);
        next_pixel = image[index2];

        if ((current_pixel == EDGE) && (next_pixel == NOT_EDGE))
        {
            edge_to_nonedge++;
        }
        if (current_pixel == EDGE)
        {
            edge_neighbors++;
        }
    }

    // Left-Up Row Clock-wise
    for (c1 = (row + 1); c1 > (row - 1); c1--)
    {
        index = (c1 * image_cols) + (col - 1);
        current_pixel = image[index];
        index2 = ((c1 - 1) * image_cols) + (col - 1);
        next_pixel = image[index2];
```

```
        if ((current_pixel == EDGE) && (next_pixel == NOT_EDGE))
        {
            edge_to_nonedge++;
        }
        if (current_pixel == EDGE)
        {
            edge_neighbors++;
        }
    }

    A = image[((row - 1) * image_cols) + col];
    B = image[(row * image_cols) + (col + 1)];
    C = image[(row * image_cols) + (col - 1)];
    D = image[((row + 1) * image_cols) + col];

    if (edge_to_nonedge == 1)
    {
        if ((edge_neighbors >= 2) && (edge_neighbors <= 6))
        {
            if ((A == NOT_EDGE) || (B == NOT_EDGE) || ((C == NOT_EDGE) && (D == NOT_EDGE)))
            {
                *mark_pixel = 1;
            }
            else
            {
                *mark_pixel = 0;
            }
        }
        else
        {
            *mark_pixel = 0;
        }
    }
    else
    {
        *mark_pixel = 0;
    }
```

```
    if (edge_to_nonedge == 1)
    {
        *end_points = 1;
    }
    if (edge_to_nonedge > 2)
    {
        *branch_points = 2;
    }
}
```

STEP 4

Determine end points and branch points and store it on a binary image

- *Function "get_end_and_branch_points()" also calls function "get_transitions()" which will return both end points and branch points and values will be used. In this case, the "marked_pixe()" variable will not be used.*
- *Function "get_end_and_branch_points()" will also create a binary image which will mark end points and branch points in the thinned image for report purposes.*

```
/* RETURNS IMAGE DETAILING IF PIXEL IS END POINT OR BRANCH POINT */
unsigned char *get_end_and_branch_points(unsigned char *image, int image_rows, int image_cols)
```

```
// Variable Declaration Section
int row = 0;
int c1 =0;
int col = 0;
int index = 0;
int mark_pixel = 0;
int end_points = 0;
int branch_points = 0;
int num_of_endp = 0;
int num_of_branchp = 0;
unsigned char *end_and_branch_point_image;
unsigned char *thinned_with_points;

// Allocate memory for image
end_and_branch_point_image = (unsigned char *)calloc(image_rows * image_cols, sizeof(unsigned char));
thinned_with_points = (unsigned char *)calloc(image_rows * image_cols, sizeof(unsigned char));

for (c1 = 0; c1 < (image_rows * image_cols); c1++)
{
    thinned_with_points[c1] = image[c1];
}

for (row = 1; row < (image_rows - 1); row++)
{
    for (col = 1; col < (image_cols - 1); col++)
    {
        index = (row * image_cols) + col;
        end_points = 0;
        branch_points = 0;
```

```
        if (image[index] == 255)
        {
            get_transitions(image, image_rows, image_cols, row, col, &mark_pixel, &end_points, &branch_points);
            if (end_points == 1)
            {
                end_and_branch_point_image[index] = 50;
                thinned_with_points[index] = 100;
                num_of_endp += 1;
            }
            if (branch_points == 2)
            {
                end_and_branch_point_image[index] = 150;
                thinned_with_points[index] = 200;
                num_of_branchp += 1;
            }
        }
    }
}
```

```
    printf("Found Endpoints: %d | Found Brachpoints: %d\n", num_of_endp, num_of_branchp);

    save_image(thinned_with_points, "thinned_with_points.ppm", image_rows, image_cols);

    return end_and_branch_point_image;
}
```

STEP 5

Threshold the MSF image with different T values, determine if letter is an “e” based on location given the ground truth text file, check if “e” is found with constraint of only one branch point and one end point, and finally save all TP, FP, TF, TN, FPR, TPN, and PPV to a CSV file.

```
void roc(unsigned char *msf_image, unsigned char *end_and_branch_point_image, int msf_rows, int msf_cols, int end_rows, int end_cols, char *file_name)
```

```
// Variable Declaration Section
FILE *file, *csv_file;
int c1, c2;
int rows, cols;
int row1, col1;
int tp, fp, fn, tn;
int threshold;
int index;
int found;
int end_points = 0;
int branch_points = 0;
char current_character[2];
char desired_character[2];
unsigned char *temp_image;
rows = cols = tp = fp = fn = tn = threshold = index = found = 0;

strcpy(desired_character, "e");

// Read in Ground Truth text file
file = fopen(file_name, "r");
if (file == NULL)
{
    printf("Error, could not read Ground Truth text file\n");
    exit(1);
}

// Allocate memory for temporary image
temp_image = (unsigned char *)calloc(msf_rows * msf_cols, sizeof(unsigned char));

// Create CSV file and write the header
csv_file = fopen("Truth Table.csv", "w");
fprintf(csv_file, "Threshold,TP,FP,FN,TN,TPR,FPR,PPV\n");
```

```
/* THRESHOLD IMAGE BASED ON THRESHOLD */
for (c1 = 0; c1 <= 250; c1 += 5)
{
    threshold = c1;

    for (c2 = 0; c2 < (msf_rows * msf_cols); c2++)
    {
        if (msf_image[c2] >= threshold)
        {
            temp_image[c2] = 255;
        }
        else
        {
            temp_image[c2] = 0;
        }
    }

    /* EXTRACT DATA REGARDING CHARACTER, COLUMNS, AND ROWS FROM GROUND TRUTH TEXT FILE */
    while((fscanf(file, "%s %d %d\n", current_character, &cols, &rows)) != EOF)
    {
        end_points = 0;
        branch_points = 0;
        found = 0;
        //cols = cols - 1;
```

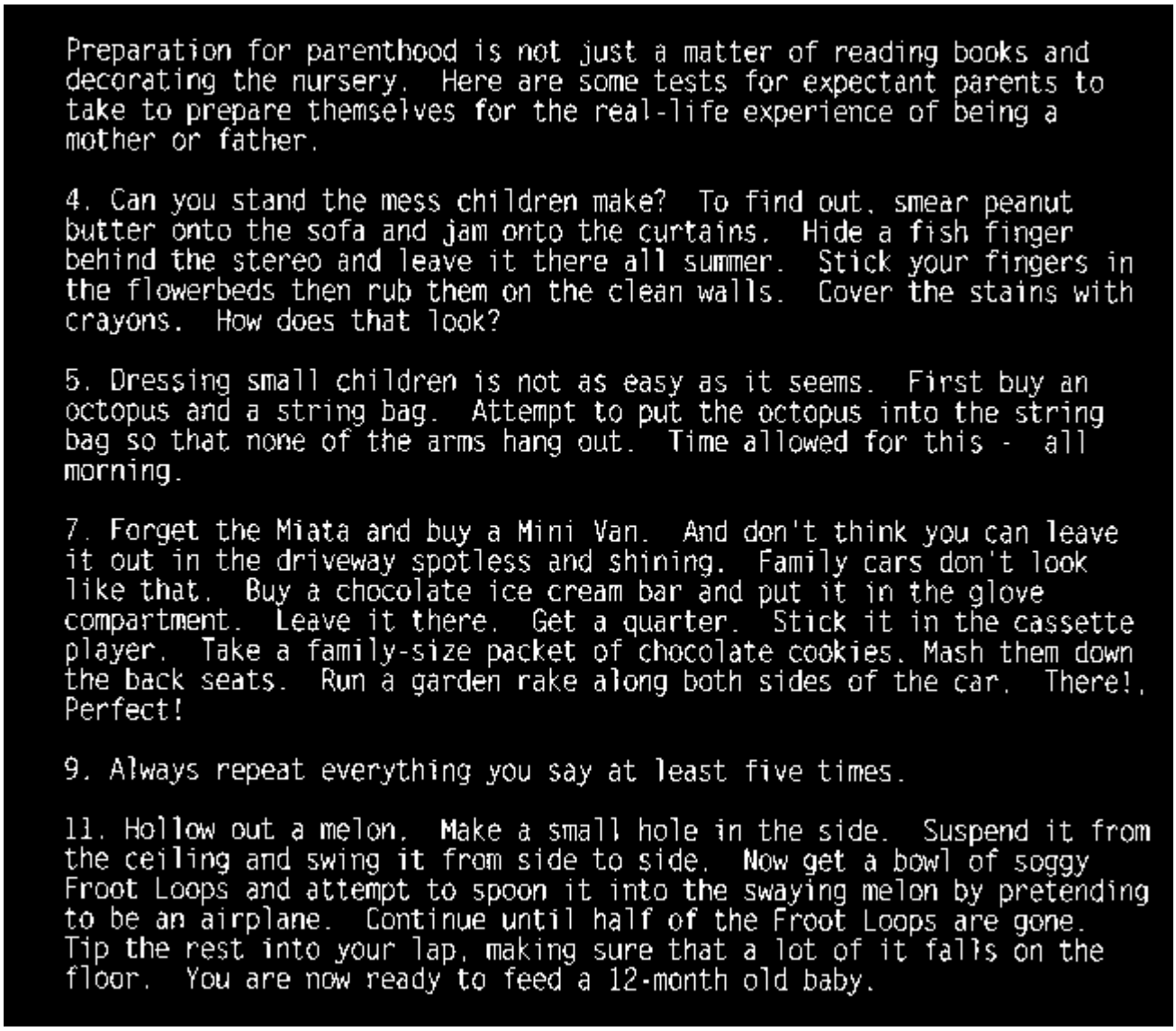
```
for (row1 = rows - 7; row1 <= (rows + 7); row1++)
{
    for (col1 = cols - 4; col1 <= (cols + 4); col1++)
    {
        index = (row1 * msf_cols) + col1;
        if (temp_image[index] == 255)
        {
            found = 1;
        }
        if (end_and_branch_point_image[index] == 50)
        {
            end_points += 1;
        }
        if (end_and_branch_point_image[index] == 150)
        {
            branch_points += 1;
        }
    }
}


if ((found == 1) && (end_points == 1) && (branch_points == 1))
{
    found = 1;
}
else
{
    found = 0;
}
```

```
        if ((found == 1) && (strcmp(current_character, desired_character) == 0))
        {
            tp++;
        }
        if ((found == 1) && (strcmp(current_character, desired_character) != 0))
        {
            fp++;
        }
        if ((found == 0) && (strcmp(current_character, desired_character) == 0))
        {
            fn++;
        }
        if ((found == 0) && (strcmp(current_character, desired_character) != 0))
        {
            tn++;
        }
    }
    // Write values to CSV file
    fprintf(csv_file, "%d,%d,%d,%d,%d,%.2f,%.2f,%.2f\n", threshold,
    tp, fp, fn, tn, tp/(double)(tp +fn ),fp/(double)(fp+tn), fp/(double)(tp+fp));
    tp = fp = fn = tn = 0;
    rewind(file);
}
```

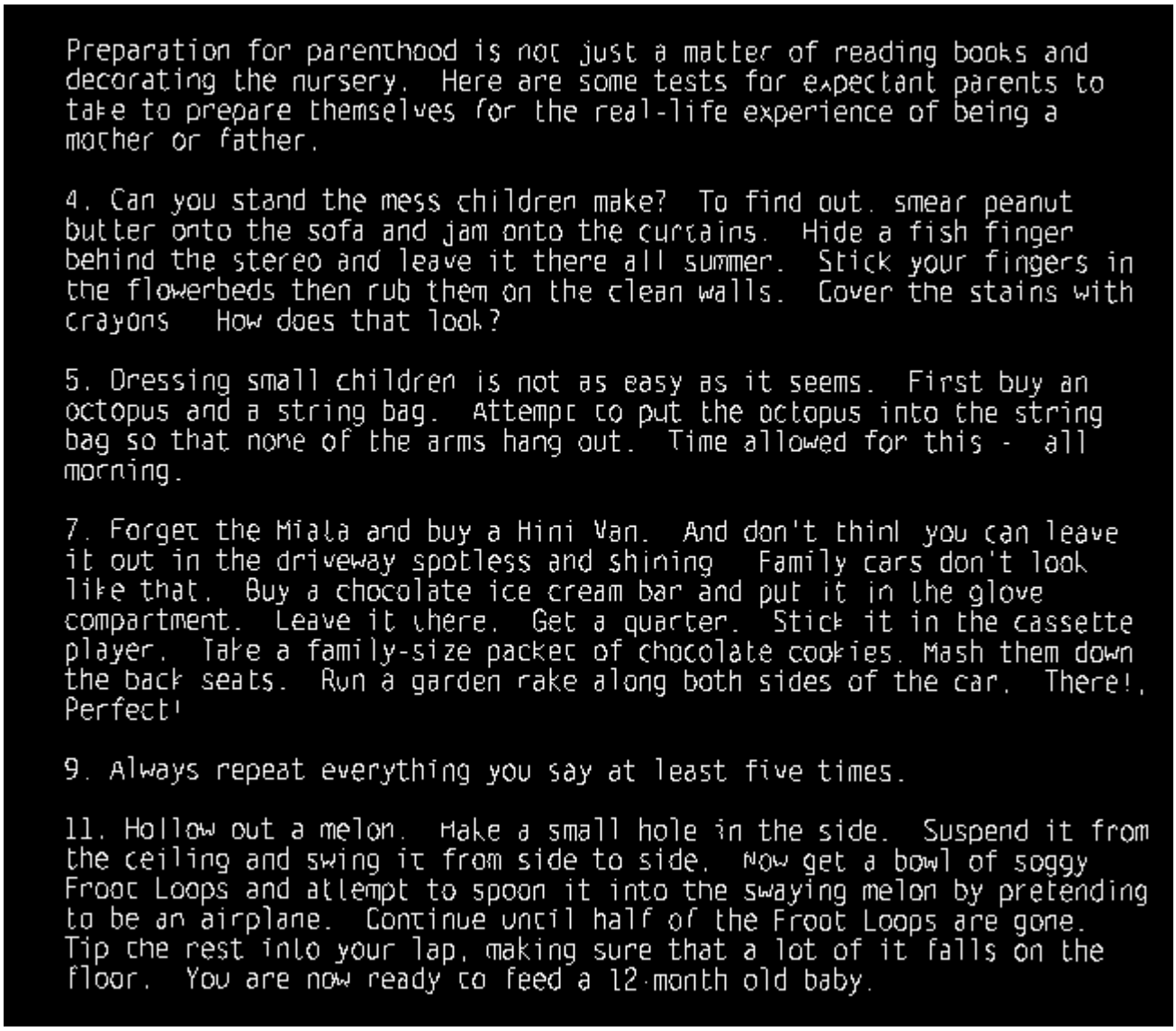
```
    fclose(file);
    fclose(csv_file);
}
```

RESULTS

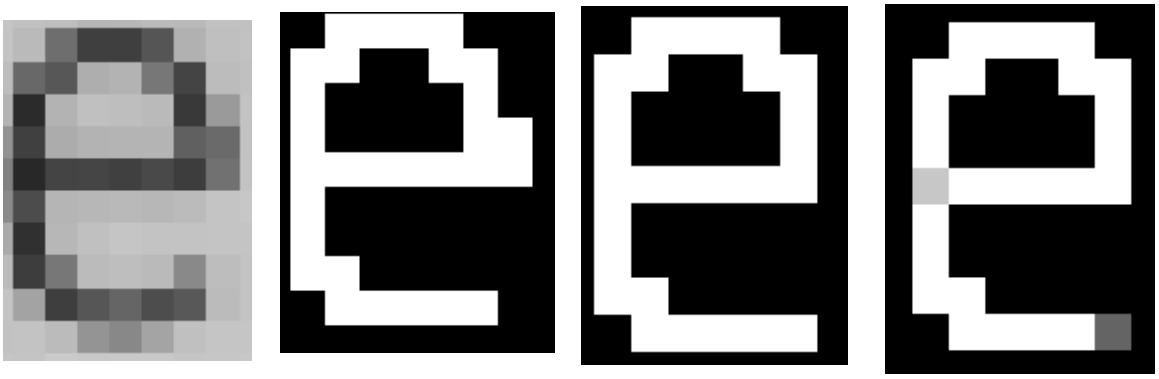
Input mage with Threshold at value 128:



Thinned Image:



Thinned image with endpoints and branch points example:

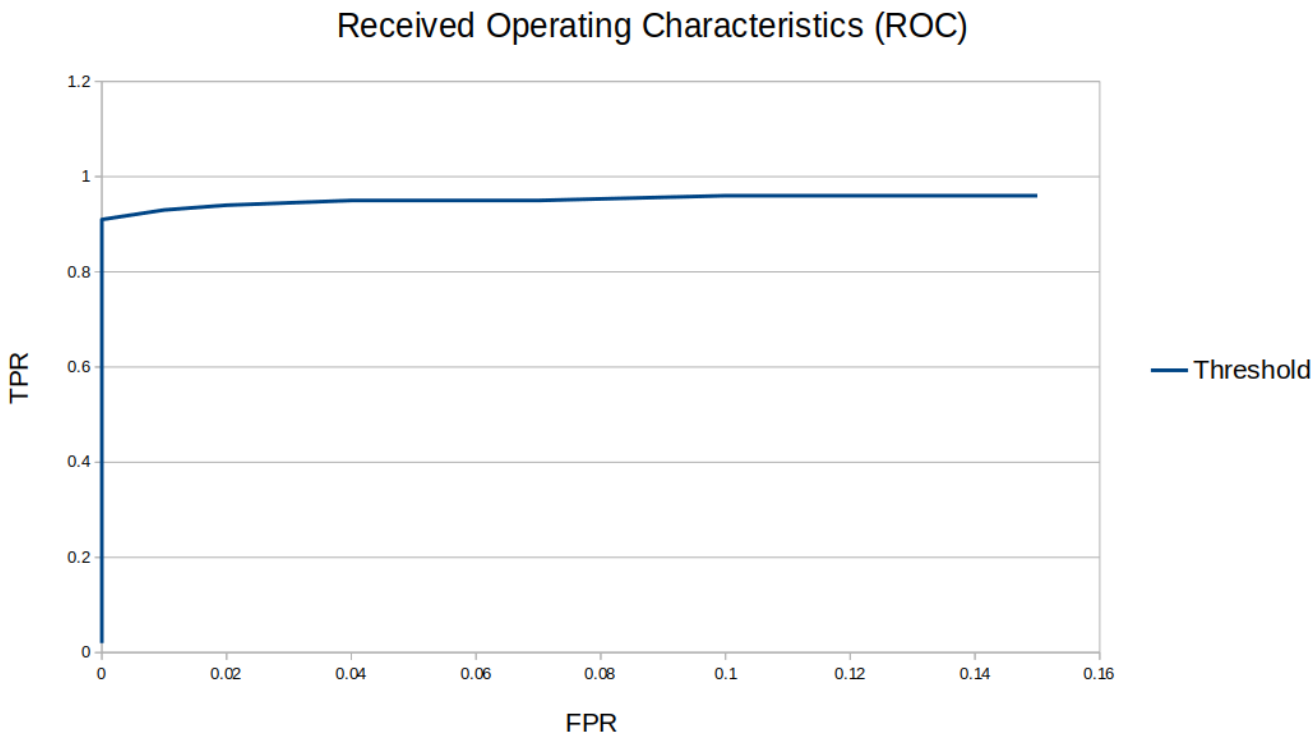


Original "e" → 128 Threshold "e" → Thinned "e" → Thinned "e" with end point and branch point

Truth Table:

Threshold	TP	FP	FN	TN	TPR	FPR	PPV
0	145	166	6	945	0.96	0.15	0.53
5	145	166	6	945	0.96	0.15	0.53
10	145	166	6	945	0.96	0.15	0.53
15	145	166	6	945	0.96	0.15	0.53
20	145	166	6	945	0.96	0.15	0.53
25	145	166	6	945	0.96	0.15	0.53
30	145	166	6	945	0.96	0.15	0.53
35	145	166	6	945	0.96	0.15	0.53
40	145	166	6	945	0.96	0.15	0.53
45	145	166	6	945	0.96	0.15	0.53
50	145	166	6	945	0.96	0.15	0.53
55	145	166	6	945	0.96	0.15	0.53
60	145	166	6	945	0.96	0.15	0.53
65	145	166	6	945	0.96	0.15	0.53
70	145	166	6	945	0.96	0.15	0.53
75	145	166	6	945	0.96	0.15	0.53
80	145	166	6	945	0.96	0.15	0.53
85	145	166	6	945	0.96	0.15	0.53
90	145	166	6	945	0.96	0.15	0.53
95	145	166	6	945	0.96	0.15	0.53
100	145	166	6	945	0.96	0.15	0.53
105	145	166	6	945	0.96	0.15	0.53
110	145	166	6	945	0.96	0.15	0.53
115	145	166	6	945	0.96	0.15	0.53
120	145	166	6	945	0.96	0.15	0.53
125	145	166	6	945	0.96	0.15	0.53
130	145	166	6	945	0.96	0.15	0.53
135	145	166	6	945	0.96	0.15	0.53
140	145	166	6	945	0.96	0.15	0.53
145	145	166	6	945	0.96	0.15	0.53
150	145	166	6	945	0.96	0.15	0.53
155	145	166	6	945	0.96	0.15	0.53
160	145	166	6	945	0.96	0.15	0.53
165	145	162	6	949	0.96	0.15	0.53
170	145	160	6	951	0.96	0.14	0.52
175	145	142	6	969	0.96	0.13	0.49
180	145	113	6	998	0.96	0.1	0.44
185	144	79	7	1032	0.95	0.07	0.35
190	143	44	8	1067	0.95	0.04	0.24
195	142	19	9	1092	0.94	0.02	0.12
200	141	9	10	1102	0.93	0.01	0.06
205	138	1	13	1110	0.91	0	0.01
210	134	0	17	1111	0.89	0	0
215	119	0	32	1111	0.79	0	0
220	104	0	47	1111	0.69	0	0
225	85	0	66	1111	0.56	0	0
230	64	0	87	1111	0.42	0	0
235	42	0	109	1111	0.28	0	0
240	30	0	121	1111	0.2	0	0
245	12	0	139	1111	0.08	0	0
250	3	0	148	1111	0.02	0	0

Received Operating Characteristics (ROC) :



Based on the ROC graph, the best thresholds in which the found/not found for letter “e” is at 200 with TP, FP, TPR, and FPR values of 141, 9, 0.93 and 0.01 correspondingly.