DSCM026 Обработка на изображения и разпознаване на образи

Изготвил: Зорница Христова

Факултетен номер: 120168

Домашна работа №2

1. **Условие на задачата:**

   Implement a adaptive Otsu's binariaztion algorithm . On each step of the algorithm, calculate the separability measure \eta. If \eta is less than a given threshold, divide the image by half, and apply the same method separately on both haves. Continue recursively. The recursion must stop either when \eta is big enough to apply Otsu's global method for this sub-image, or when the sub-image is too small.

Решение на задачата:

Introduction to the Otsu Binarization:

Otsu Binarization is an automatic thresholding method used to convert grayscale images into binary (black and white) images. Unlike manual thresholding, Otsu's method automatically determines the optimal threshold value by analyzing the image's histogram.

<u>**Key words:**</u> ***Grayscale Image:*** *Image with pixel values ranging from 0 (black) to 255 (white);* ***Binary Image****: image with only two values: 0 (black) and 255 (white);* ***Threshold****: a value that separates pixels into two groups (foreground and background)****); Histogram****: a graph showing the frequency of each pixel intensity value;*

<u>**Algorithm Mathematical Overview:**</u> Let the pixels of a given picture be represented in L gray levels [1, 2, ..., L]. The number of pixels at level i is denoted by n_i and the total number of pixels by N = n_1 + n_2 + ... + n_L. In order to simplify the discussion, the gray-level histogram is normalized and regarded as a probability distribution:

$$(1)\quad p_i = n_i / N,\ \ p_i \geq 0,\ \ \Sigma(i=1 \text{ to } L)\ p_i = 1$$

where:

- p_i represents the probability of a pixel having gray level i
- n_i is the number of pixels at gray level i
- N is the total number of pixels in the image
- L is the total number of gray levels (typically 256 for 8-bit images)

The Otsu method aims to find an optimal threshold value k that separates the image into two classes:

- Class C_0: pixels with gray levels [1, 2, ..., k] (background)
- Class C_1: pixels with gray levels [k+1, k+2, ..., L] (foreground)

Hence, the probabilities of the two classes are:

$$(2)\ \omega_0(k) = \Sigma(i=1 \text{ to } k)\ p_i$$

(3) $\omega_1(k) = \Sigma(i=k+1 \text{ to } L)\ p_i = 1 - \omega_0(k)$

The mean gray levels of each class are calculated as:

(4) $\mu_0(k) = \Sigma(i=1 \text{ to } k)\ i \times p_i / \omega_0(k)$ - the mean intensity of class C_0 (background)

(5) $\mu_1(k) = \Sigma(i=k+1 \text{ to } L)\ i \times p_i / \omega_1(k)$ - the mean intensity of class C_1 (foreground)

The global mean intensity of the entire image is:

(6) $\mu_T = \Sigma(i=1 \text{ to } L)\ i \times p_i$

The following formula measures how well the threshold k separates the two classes. A larger between-class variance indicates better separation between foreground and background:

(7) $\sigma^2_B(k) = \omega_0(k) \times \omega_1(k) \times [\mu_0(k) - \mu_1(k)]^2$

Hence, the Otsu method finds the optimal threshold k* by maximizing the between-class variance:

(8) $k^* = \text{argmax}(k)\ \sigma^2_B(k)$, for $k = 1, 2, \ldots, L-1$

The goal is to select the threshold k that maximizes the separation between these two classes.

Finally, when k* is found, apply the threshold: pixels with level ≤ k* become 0 (black), pixels with level > k* become L (white).

Code Implementation in C++:

Define the global variable needed:

```
INTENS_MIN = 0;   // darkest intensity possible

INTENS_MAX = 255; // brightest intensity possible

const double EPS = 1.0e-14;

const double ETA_THRESHOLD = 0.5; //Threshold for separability measure

const int MIN_IMAGE_SIZE = 32;     // Minimum size to continue recursion
```

Explanation of the need for the above global constants:
//ETA_THRESHOLD: If η (eta) ≥ 0.5, the region is considered "separable enough" to apply Otsu directly

//MIN_IMAGE_SIZE: If a sub-image is smaller than 32×32 pixels, stop recursion and apply Otsu directly

```
const QString file_name = "gray_lenna.png"
```

We start with computing the normalized histogram (probability distribution) of pixel intensities for a given rectangular region:

```
void calcHisto(QImage &image, int x1, int y1, int x2, int y2, double histo[])
{
    for (int i = 0; i <= INTENS_MAX; i++)
    {
        histo[i] = 0;
    }
    int numb_pix = 0;
    for (int indx_row = y1; indx_row < y2; indx_row++)
    {
        if (indx_row >= 0 && indx_row < image.height())
        {
            quint8* ptr_row = (quint8*)(image.bits()
                    + indx_row * image.bytesPerLine());
            for (int indx_col = x1; indx_col < x2; indx_col++)
            {
                if (indx_col >= 0 && indx_col < image.width())
                {
                    histo[ptr_row[indx_col]]++;
                    numb_pix++;
                }
            }
        }
    }
    if (numb_pix > 0)
    {
        for (int i = 0; i <= INTENS_MAX; i++)
        {
            histo[i] /= numb_pix;
```

Then we compute the first-order cumulative moments of the histogram up to the INTENS_MAX level (total mean level of the picture) and then the total variance which uses the global mean with the function **calculateTotalVariance:**

```
double calculateTotalVariance(const double histo[])
{
    // First calculate the global mean
    double m_g = 0.0;
    for (int i = 0; i <= INTENS_MAX; i++)
    {
        m_g += i * histo[i];
    }


    // Then calculates total variance which uses the global mean σ²_T = Σ(i − μ_T)² ×
p_i
    double sigma_T = 0.0;
    for (int i = 0; i <= INTENS_MAX; i++)
    {
        double diff = i - m_g;
        sigma_T += diff * diff * histo[i];
    }


    return sigma_T;
}
```

And finally we implement the main recursive algorithm: implements the adaptive Otsu algorithm with recursive division

The algorithm flow is the following:

```
void applyThreshold(QImage &image, int x1, int y1, int x2, int y2, int threshold)
{
    for (int indx_row = y1; indx_row < y2; indx_row++)
    {
        if (indx_row >= 0 && indx_row < image.height())
        {
            quint8* ptr_row = (quint8*)(image.bits()
                    + indx_row * image.bytesPerLine());
            for (int indx_col = x1; indx_col < x2; indx_col++)
            {
                if (indx_col >= 0 && indx_col < image.width())
                {
                    ptr_row[indx_col] =
                        (ptr_row[indx_col] > threshold) ? INTENS_MAX : INTENS_MIN;
                }
            }
        }
    }
}
```

Now we calculate Separability Measure η

The formula is: $\eta = \sigma^2\_B / \sigma^2\_T$

- `$\sigma^2\_B$` = between-class variance (how well the threshold separates classes)

- `$\sigma^2\_T$` = total variance (overall spread of intensities)

```
double calculateEta(const double histo[], int threshold)
{
    // Calculate cumulative sums
    double p_1[INTENS_MAX + 1] = {0};
    p_1[0] = histo[0];
    for (int i = 1; i <= INTENS_MAX; i++)
    {
          //Calculates cumulative probability `p_1[i]` = probability of intensity ≤ i
        p_1[i] = p_1[i - 1] + histo[i];
    }

    // Cumulative mean
    double m[INTENS_MAX + 1] = {0};
    for (int i = 1; i <= INTENS_MAX; i++)
    {
          //Calculates cumulative mean `m[i]` = mean intensity up to level i
        m[i] = m[i - 1] + i * histo[i];
    }

    // Global mean
    double m_g = m[INTENS_MAX];

    // Between-class variance for given threshold
    double div = (p_1[threshold] * (1 - p_1[threshold]));
    double sigma_B = 0.0;
    if (fabs(div) >= EPS)
    {
          //σ²_B = (m_g × p_1[t] - m[t])² / (p_1[t] × (1 - p_1[t]))
        double diff = m_g * p_1[threshold] - m[threshold];
        sigma_B = (diff * diff) / div;
    }

    // Total variance
    double sigma_T = calculateTotalVariance(histo);

    // Eta = σ²_B / σ²_T
    if (fabs(sigma_T) < EPS)
    {
        return 0.0;
    }

    return sigma_B / sigma_T;
}    {
        double diff = i - m_g;
        sigma_T += diff * diff * histo[i];
    }

    return sigma_T;
}
```

Now we use **otsu** function to find an optimal threshold which maximizes between-class variance:
It calculates cumulative sums and means, for each possible threshold `i` (0-255), computes between-class variance, returns the threshold with maximum between-class variance.

```
int otsu(const double histo[])
{
    // compute cumulative sums
    double p_1[INTENS_MAX + 1] = {0};
    p_1[0] = histo[0];
    for (int i = 1; i <= INTENS_MAX; i++)
    {
        p_1[i] = p_1[i - 1] + histo[i];
    }

    // cumulative mean
    double m[INTENS_MAX + 1] = {0};
    for (int i = 1; i <= INTENS_MAX; i++)
    {
        m[i] = m[i - 1] + i * histo[i];
    }

    // global mean
    double m_g = m[INTENS_MAX];
```

Then we apply binary threshold with the function **applyThreshold.** This function sets each pixel in the region `(x1, y1)` to `(x2, y2)` to:
  - If pixel intensity > threshold → set to 255 (white)
  - If pixel intensity ≤ threshold → set to 0 (black)

The result is a binary image in that region.

```
void applyThreshold(QImage &image, int x1, int y1, int x2, int y2, int threshold)
{
    for (int indx_row = y1; indx_row < y2; indx_row++)
    {
        if (indx_row >= 0 && indx_row < image.height())
        {
            quint8* ptr_row = (quint8*)(image.bits()
                    + indx_row * image.bytesPerLine());
            for (int indx_col = x1; indx_col < x2; indx_col++)
            {
                if (indx_col >= 0 && indx_col < image.width())
                {
                    ptr_row[indx_col] =
                        (ptr_row[indx_col] > threshold) ? INTENS_MAX : INTENS_MIN;
                }
            }
        }
    }
}
    // between-class
```

Finally we apply the Main Recursive Algorithm with the function **adaptiveOtsu.** The function works in the following way**:**

1. Check if sub-image is too small (< MIN_IMAGE_SIZE)

→ YES: Apply Otsu directly, return

→ NO: Continue

2. Calculate histogram for current region

3. Find optimal threshold using Otsu's method

4. Calculate separability measure η for that threshold

5. Check if η ≥ ETA_THRESHOLD

→ YES: Region is separable enough

→ Apply threshold directly

→ Return (stop recursion)

→ NO: Region is not separable enough

→ Divide region in half:

- If width > height: split vertically (divide width)

- If height ≥ width: split horizontally (divide height)

→ Recursively call adaptiveOtsu() on both halves

→ Continue until η ≥ threshold OR region too small

```
void adaptiveOtsu(QImage &image, int x1, int y1, int x2, int y2, int depth = 0)

{

    int width = x2 - x1;

    int height = y2 - y1;


    // Check if sub-image is too small

    if (width < MIN_IMAGE_SIZE || height < MIN_IMAGE_SIZE)

    {

        QTextStream(stdout) << "Sub-image too small (" << width << "x" << height

                           << "), applying Otsu directly" << Qt::endl;

        double histo[INTENS_MAX + 1];

        calcHisto(image, x1, y1, x2, y2, histo);

        int th = otsu(histo);

        applyThreshold(image, x1, y1, x2, y2, th);

        return;

    }


    // Calculate histogram for this sub-image

    double histo[INTENS_MAX + 1];

    calcHisto(image, x1, y1, x2, y2, histo);


    // Find optimal threshold

    int optimal_th = otsu(histo);


    // Calculate separability measure eta

    double eta = calculateEta(histo, optimal_th);


    QTextStream(stdout) << "Depth " << depth << ": Region [" << x1 << "," << y1

                       << "] to [" << x2 << "," << y2 << "] - Eta: " << eta

                       << ", Threshold: " << optimal_th << Qt::endl;


    // If eta is big enough, apply threshold directly

    if (eta >= ETA_THRESHOLD)

    {

        QTextStream(stdout) << "Eta >= threshold, applying Otsu directly" << Qt::endl;

        applyThreshold(image, x1, y1, x2, y2, optimal_th);

        return;

    }


    // Otherwise, divide image and recurse

    QTextStream(stdout) << "Eta < threshold, dividing image..." << Qt::endl;


    // Divide horizontally or vertically (choose larger dimension)
```

Results: an output example for the input picture defined with const QString file_name = "gray_lenna.png";

Image loaded: gray_lenna.png

```
Format: 24
Size: 512x512
Starting adaptive Otsu binarization...
Eta threshold: 0.5
Min image size: 32
Depth 0: Region [0,0] to [512,512] - Eta: 0.699222, Threshold: 116
Eta >= threshold, applying Otsu directly
Adaptive Otsu binarization completed!
```