UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

License Plate Detection

Procesarea Imaginilor

Autori: Vlase Rafaella si Pescaru Silviu
Grupa: 30235

Facultatea de Automatica
si Calculatoare

24 Mai 2024

Cuprins

1 Tema proiectului

Detecția frontală, din imagini (color), a plăcuțelor cu numere de înmatriculare de România ale autovehiculelor.

- se dau imagini (color) cu autovehicule în care numărul de înmatriculare apare în poziție frontală
- algoritmul trebuie să identifice și să marcheze zona în care se află acesta cu un bounding box
- pentru 2 studenți algoritmul trebuie să includă și transformarea numărului de înmatriculare detectat în text

2 Specificatii detaliate

2.1 Formatul datelor de intrare si de iesire

2.1.1 Date de intrare:

Imagini frontale cu autovehicule, in care se vad placutele cu numere de inmatriculare de Romania

2.1.2 Date de iesire:

- Imaginea cropped doar cu placuta de inmatriculare
- Imaginea in format edged dupa aplicarea algoritmului Canny de detectie a muchiilor
- Imaginea originala pentru referinta
- Textul de pe placuta de inmatriculare afisat in consola

2.2 Prezentarea temei

Se cere un program implementat in C++, utilizand biblioteca OpenCV, care urmareste detectarea numerelor de pe placutele de inmatriculare din Romania. Aceasta tema presupune atat prezentarea unor imagini intermediare ale pasilor si etapelor prin care trece imaginea sursa, cat si valorificarea rezultatului final, afisarea in consola a numarului de inmatriculare extras de pe placuta.

2.3 Cerinte functionale

- Implementarea algoritmului Gaussian Blur
- Implementarea algoritmului Canny pentru detectia muchiilor
- Implementarea altor algoritmi auxiliari
- Utilizarea bibliotecii OpenCV si a altor biblioteci necesare pentru detectia textului (e.g. Tesseract OCR)

3 Analiza si fundamentare teoretica

3.1 Algoritmi utilizati

3.1.1 Algoritmul Gaussian Blur

Am implementat acest algoritm folosindu-ne de doua functii aditionale, una pentru operatia de convolutie si una pentru calculul scalar.

Operația de convoluție implică folosirea unei măști/nucleu de convoluție H care se aplică peste imaginea sursă. Filtrul Gaussian este un filtru de netezire utilizat frecvent în procesarea imaginilor pentru a reduce zgomotul.

3.1.2 Functia connectEdges

Funcția connectEdges este concepută pentru a conecta marginile într-o imagine, care este o etapă comună în procesarea imaginilor, în special în detectarea marginilor.

3.1.3 Algoritmul Canny pentru detectia muchiilor

Algoritmul Canny pentru detectarea muchiilor este un operator de detectare a marginilor care utilizează un algoritm în mai multe etape pentru a detecta o gamă largă de margini în imagini. Algoritmul este compus din 5 pasi:

- Reducerea zgomotului
- Calculul gradientului
- Suprimarea non-maximelor
- Binarizare adaptativa
- Extinderea muchiilor prin histereza

3.1.4 Algoritm pentru extragerea regiunii de interes si a textului

Pentru decuparea placutei si afisarea numarului de inmatriculare am trecut prin mai multe etape:

- Detectarea contururilor
- Sortarea contururilor
- Extragerea regiunii de interes
- Binarizarea imaginii
- Extragerea textului cu OCR (Optical Character Recognition)

3.2 Argumentarea solutiei alese

Am ales aceasta implementare deoarece algoritmul Canny este unul dintre cei mai eficienti algoritmi de detectie a muchiilor. In plus, algoritmul de sortare a contururilor și algoritmul de aproximare a contururilor permit identificarea precisă a plăcuțelor de înmatriculare într-o imagine.

4 Proiectare de detaliu si implementare

4.1 Schema bloc

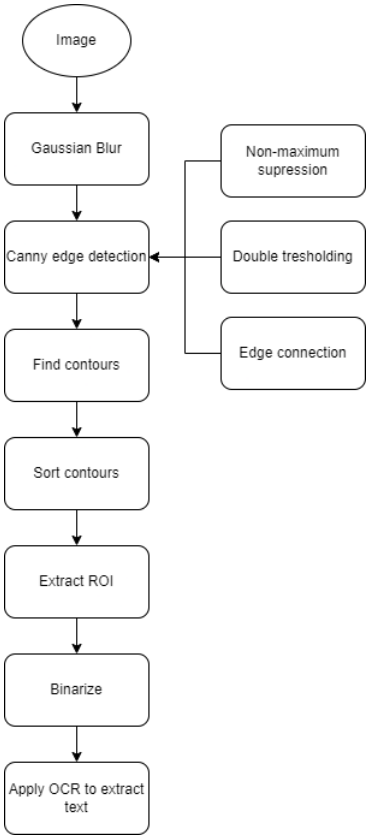

Figura 1: Schema bloc a proiectului

4.2 Descrierea componentelor

4.2.1 Filtrul Gaussian

Filtrul Gaussian este definit de o matrice de kernel, care în acest caz este o matrice 3x3. Aceasta este o matrice de ponderi, unde fiecare valoare reprezintă ponderea unui pixel vecin atunci când se calculează noua valoare a pixelului central.

Funcția calculScalar calculează scalarul pentru normalizarea valorilor pixelilor după aplicarea filtrului. În acest caz, scalarul este maximul dintre suma valorilor pozitive și suma valorilor absolute ale valorilor negative din kernel.

```
float calculScalar(int Kernel[3][3]) {
    int sum1 = 0;
    int sum2 = 0;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (Kernel[i][j] < 0) {
                sum1 += abs(Kernel[i][j]);
            }
            else {
```

```
                sum2 += Kernel[i][j];
            }
        }
    }

    float scalar = max(sum1, sum2);
    return scalar;
}
```

Funcția convolution aplică filtrul asupra imaginii de intrare (src). Pentru fiecare pixel din imagine, se calculează suma ponderată a valorilor pixelilor vecini folosind kernelul, iar rezultatul este împărțit la scalar pentru normalizare. Rezultatul este stocat în imaginea de ieșire (dst).

```
float calculScalar(int Kernel[3][3]) {
    int sum1 = 0;
    int sum2 = 0;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (Kernel[i][j] < 0) {
                sum1 += abs(Kernel[i][j]);
            }
            else {
                sum2 += Kernel[i][j];
            }
        }
    }

    float scalar = max(sum1, sum2);
    return scalar;
}
```

Funcția filtruGaussian este funcția principală care inițializează kernelul Gaussian, calculează scalarul și apoi aplică operația de convoluție asupra imaginii de intrare. Rezultatul este o imagine netezită, unde detaliile fine și zgomotul au fost reduse.

```
Mat filtruGaussian(Mat src) {
    int Kernel[3][3] = { {1, 2, 1},
                         {2, 4, 2},
                         {1, 2, 1} };
    float scalar = calculScalar(Kernel);
    Mat dst = convolution(src, Kernel, scalar);
    return dst;
}
```

4.2.2 Conectarea muchiilor

Funcția connectEdges este concepută pentru a conecta marginile într-o imagine. Funcția primește o imagine de intrare (src) și creează o copie a acesteia (dst) pe care o va modifica.

Pentru fiecare pixel din imagine, dacă valoarea acestuia este 255 (ceea ce înseamnă că este un pixel de margine), funcția începe un proces de căutare în lățime pentru a găsi și a conecta toți pixelii de margine vecini. Acest lucru se face prin adăugarea vecinilor pixelului curent în coadă și setarea valorii lor la 255 în imaginea de ieșire.

După ce toate marginile conectate au fost găsite și marcate, funcția parcurge din nou imaginea și setează toți pixelii cu valoarea 128 la 0. Acest lucru se face pentru a elimina pixelii care au fost inițial considerați ca făcând parte dintr-o margine, dar care nu au fost conectați cu nicio margine în procesul BFS. Rezultatul final este o imagine în care toate marginile au fost conectate.

```
Mat connectEdges(Mat src) {
    int height = src.rows;
    int width = src.cols;

    int di[] = { 0, -1, -1, -1, 0, 1, 1, 1 };
    int dj[] = { 1, 1, 0, -1, -1, -1, 0, 1 };

    Mat dst = src.clone();

    for (int i = 1; i < height - 1; i++) {
        std::cout << "Connecting edges " << (i * 100) / height << "%\n";
        for (int j = 1; j < width - 1; j++) {
            if (dst.at<float>(i, j) == 255) {
                queue<pair<int, int>> q;
                q.push(make_pair(i, j));
                while (!q.empty()) {
                    pair<int, int> punct = q.front();
                    q.pop();
                    for (int k = 0; k < 8; k++) {
                        if (punct.first + di[k] < height && punct.second + dj[k] < width) {
                            if (dst.at<float>(punct.first + di[k], punct.second + dj[k])
                            == 128)
                            {
                                dst.at<float>(punct.first + di[k], punct.second
                                + dj[k]) = 255;
                                q.push(make_pair(punct.first + di[k], punct.second
                                + dj[k]));
                            }
                        }
                    }
                }
            }
        }
    }

    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (dst.at<float>(i, j) == 128) {
                dst.at<float>(i, j) = 0;
```

```
            }
        }
    }
    return dst;
}
```

4.2.3 Algoritmul Canny pentru detectia muchiilor

Algoritmul începe prin aplicarea unui filtru Gaussian asupra imaginii de intrare pentru a reduce zgomotul. Apoi, algoritmul calculează gradientul imaginii folosind operatorul Sobel. Acesta calculează gradientul pe axele x și y, și apoi convertește aceste două componente în magnitudinea și unghiul gradientului. Algoritmul parcurge apoi fiecare pixel din imagine și verifică dacă este un maxim local în direcția gradientului. Dacă nu este, valoarea sa este setată la zero. Acest procedeu poarta denumirea de "suprimare non-maxime". Algoritmul aplică apoi un double tresholding pentru a clasifica pixelii ca fiind puternici, slabi sau non-margine. Pixelii cu o magnitudine a gradientului mai mare decât pragul puternic sunt considerați pixeli de margine puternici, pixelii cu o magnitudine a gradientului între pragurile puternic și slab sunt considerați pixeli de margine slabi, iar ceilalți pixeli sunt considerați pixeli non-margine. În final, algoritmul conectează marginile slabe la marginile puternice dacă sunt conectate, altfel le elimină.

Rezultatul final este o imagine "edged", în care marginile au fost detectate.

```
cv::Mat canny(cv::Mat src, double weak_th = -1, double strong_th = -1) {
    src = filtruGaussian(src.clone());
    std::cout << "Gaussian filter - finished" << std::endl;
    //compute gradients
    cv::Mat gx, gy;
    cv::Sobel(src, gx, CV_64F, 1, 0, 3);
    cv::Sobel(src, gy, CV_64F, 0, 1, 3);

    cv::Mat mag, ang;
    cv::cartToPolar(gx, gy, mag, ang, true);

    // setting the minimum and maximum thresholds for double thresholding
    double minVal, maxVal;
    cv::minMaxLoc(mag, &minVal, &maxVal);
    double mag_max = maxVal;
    if (weak_th == -1) weak_th = mag_max * 0.1;
    if (strong_th == -1) strong_th = mag_max * 0.5;

    for (int i_x = 0; i_x < src.cols; ++i_x) {
        for (int i_y = 0; i_y < src.rows; ++i_y) {

            double grad_ang = ang.at<double>(i_y, i_x);
            if (abs(grad_ang) > 180)
                grad_ang = abs(grad_ang - 180);
            else
                grad_ang = abs(grad_ang);

```

```
            // selecting the neighbours of the target pixel
            // according to the gradient direction
            // In the x axis direction
            int neighb_1_x, neighb_1_y, neighb_2_x, neighb_2_y;
            if (grad_ang <= 22.5) {
                neighb_1_x = i_x - 1; neighb_1_y = i_y;
                neighb_2_x = i_x + 1; neighb_2_y = i_y;
            }
            // top right (diagonal-1) direction
            else if (grad_ang > 22.5 && grad_ang <= (22.5 + 45)) {
                neighb_1_x = i_x - 1; neighb_1_y = i_y - 1;
                neighb_2_x = i_x + 1; neighb_2_y = i_y + 1;
            }
            // In y-axis direction
            else if (grad_ang > (22.5 + 45) && grad_ang <= (22.5 + 90)) {
                neighb_1_x = i_x; neighb_1_y = i_y - 1;
                neighb_2_x = i_x; neighb_2_y = i_y + 1;
            }
            // top left (diagonal-2) direction
            else if (grad_ang > (22.5 + 90) && grad_ang <= (22.5 + 135)) {
                neighb_1_x = i_x - 1; neighb_1_y = i_y + 1;
                neighb_2_x = i_x + 1; neighb_2_y = i_y - 1;
            }
            // Now it restarts the cycle
            else if (grad_ang > (22.5 + 135) && grad_ang <= (22.5 + 180)) {
                neighb_1_x = i_x - 1; neighb_1_y = i_y;
                neighb_2_x = i_x + 1; neighb_2_y = i_y;
            }

            // Non-maximum suppression step
            if (src.cols > neighb_1_x >= 0 && src.rows > neighb_1_y >= 0) {
                if (mag.at<double>(i_y, i_x) < mag.at<double>(neighb_1_y, neighb_1_x)) {
                    mag.at<double>(i_y, i_x) = 0;
                    continue;
                }
            }

            if (src.cols > neighb_2_x >= 0 && src.rows > neighb_2_y >= 0) {
                if (mag.at<double>(i_y, i_x) < mag.at<double>(neighb_2_y, neighb_2_x)) {
                    mag.at<double>(i_y, i_x) = 0;
                }
            }
        }
    }

    std::cout << "Non-maximum suppresion - finished" << std::endl;

    // double thrsholding step
```

```
    for (int i_y = 0; i_y < mag.rows; ++i_y) {
        for (int i_x = 0; i_x < mag.cols; ++i_x) {

            if (mag.at<double>(i_y, i_x) < weak_th) {
                mag.at<double>(i_y, i_x) = 0;
            }
            else if (strong_th > mag.at<double>(i_y, i_x) >= weak_th) {
                mag.at<double>(i_y, i_x) = 128;
            }
            else {
                mag.at<double>(i_y, i_x) = 255;
            }
        }
    }

    std::cout << "Double-tresholding - finished" << std::endl;
    // finally returning the magnitude of
    // gradients of edges
    mag = connectEdges(mag.clone());
    std::cout << "Connecting edges - finished" << std::endl;
    return mag;
}
```

4.2.4 Extragerea regiunii de interes si a numarului de inmatriculare

Algoritmul începe prin găsirea contururilor în imaginea de intrare (edged). Acest lucru se face folosind funcția findContours, care identifică contururile din imagine. Contururile sunt apoi sortate în funcție de aria lor, folosind o funcție comparator compareContourAreas. Algoritmul parcurge apoi fiecare contur și îl aproximează la o formă simplă. Dacă un contur poate fi aproximat la un poligon cu patru laturi, acesta este selectat drept conturul plăcii de înmatriculare (screenCnt). Odată ce conturul plăcii de înmatriculare a fost identificat, regiunea de interes (ROI) este extrasă din imaginea originală (gray) și este stocată în Cropped.

În final, se utilizează biblioteca Tesseract pentru a citi textul de pe placa de înmatriculare, prin Optical Character Recognition. Textul recunoscut este apoi afisat in consola.

```
vector<vector<Point>> contours;
findContours(edged.clone(), contours, RETR_TREE, CHAIN_APPROX_SIMPLE);

// Utilize the comparator function for sorting contours
sort(contours.begin(), contours.end(), compareContourAreas);

vector<Point> screenCnt;
for (auto& c : contours) {
    double peri = arcLength(c, true);
    vector<Point> approx;
    approxPolyDP(c, approx, 0.018 * peri, true);

    Rect rect = boundingRect(approx);
    double aspect_ratio = static_cast<double>(rect.width) / rect.height;
```

```
    if (approx.size() == 4) {
        cout << aspect_ratio << endl;
        screenCnt = approx;
        break;
    }
}

// Extract license plate region
Rect roi = boundingRect(screenCnt);
Mat Cropped = gray(roi);

Mat binarized;
threshold(Cropped, binarized, 0, 255, THRESH_BINARY_INV + THRESH_OTSU);

// Initialize Tesseract OCR
tesseract::TessBaseAPI* api = new tesseract::TessBaseAPI();
api->Init("C:/Users/ella/vcpkg/packages/tesseract_x64-windows-static/share/tessdata",
"eng", tesseract::OEM_DEFAULT);
api->SetImage((uchar*)binarized.data, binarized.cols, binarized.rows, 1, binarized.step);

// Get the text from the license plate
char* outText = api->GetUTF8Text();
cout << "License plate text: " << outText << endl;
```

5 Testare si validare

Mai jos avem imagini cu pasii prin care trece imaginea sursa pentru a obtine la final placuta si numarul de inmatriculare. Algoritmul prezentat este valid si respecta toate cerintele de implementare.

Figura 2: Imaginea sursa

Figura 3: Imaginea dupa aplicarea algoritmului Canny

Figura 4: Placuta de inmatriculare

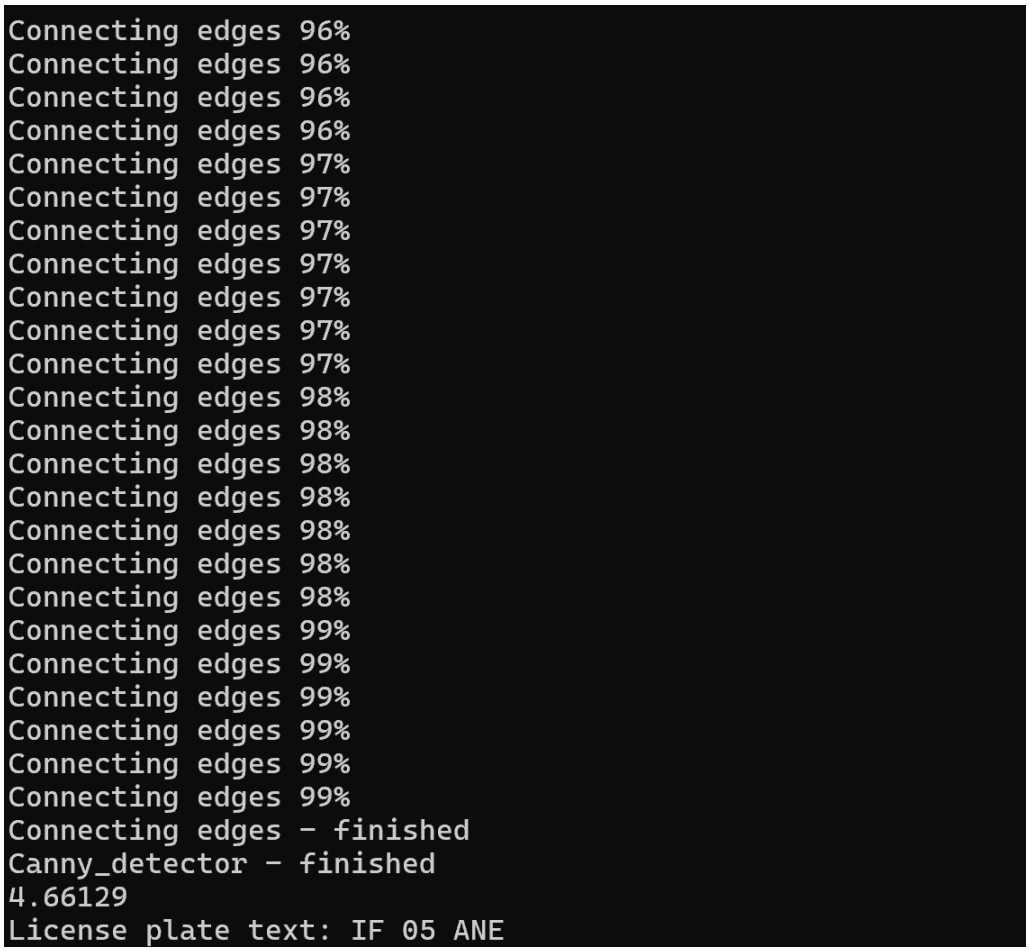

Figura 5: Afisarea textului in consola

6 Dezvoltari ulterioare

O posibila dezvoltare ulterioara poate consta in detectarea placutei de inmatriculare dintr-o imagine care nu este frontala.