Visão por Computador 2016-17, Guia Prático N.º 10

Rui Oliveira, Tomás Rodrigues
DETI, Universidade de Aveiro
Aveiro, Portugal
{ruipedrooliveira, tomasrodrigues}@ua.pt

Resumo –
Pretende-se através deste relatório expor sob forma escrita, o nosso desempenho e objetivos alcançados na aula prática n.º10 da unidade curricular de Visão por Computador do Mestrado Integrado de Engenharia de Computadores e Telemática.
Neste relatório pretenderemos explicar as soluções por nós encontradas para a resolução dos diferentes problemas propostos.

***Palavras chave* – visão, computador, imagem digital, template matching, face detector opencv, c++,**

I. REPOSITÓRIO: CÓDIGO FONTE

Todas as soluções dos problemas propostos estão disponível através do seguinte repositório (gitHub) criado para o efeito.

http://github.com/toomyy94/CV1617-68779-68129

A resolução dos problemas do presente guia encontram-se na pasta aula10. Para a resolução dos exercícios foi utilizado o CodeBlocks IDE. .

II. PROBLEMAS PROPOSTOS

A. *Problema #1*

A.1 Enunciado

Develop a program to perform traffic sign detection. You can consider the existence of a database with images of several traffic signs (templates) and your program has to find the position of the signs in the input image (or video) if they appear there. As starting point, explore the OpenCv tutorial about template matching. In your implementation, explore the several matching methods available in OpenCV.

A.2 Resolução do problema

No geral pensamos ter conseguido bons resultados na resolução do problema proposto(identificação de um sinal de trânsito em imagens).
Para a resolução deste problema numa 1ª fase criamos a matriz resultante da imagem a procurar e da imagem *template*:

Listing 1: Create the result matrix

```
int result_cols = img.cols - templ.cols + 1;
int result_rows = img.rows - templ.rows + 1;
result.create( result_rows, result_cols,
    CV_32FC1 );
```

Depois disso vai-se fazendo deslizar a imagem template na original calculando possíveis matchs ou probabilidades de match:

Listing 2: Matching and Normalize

```
matchTemplate( img, templ, result,
    match_method );
normalize( result, result, 0, 1, NORM_MINMAX,
    -1, Mat() );
```

São usados 5 algoritmos para o *SQDIFF* e o *SQDIFF_NORMED*, os melhores casos são os valores calculados correspondentes aos mais baixos, para os outros algoritmos os melhores valores, são os valores mais altos.

Listing 3: Desenho de retangulos no ponto de interesse

```
if( match_method == CV_TM_SQDIFF || match_method
    == CV_TM_SQDIFF_NORMED )
  { matchLoc = minLoc; }
 else
  { matchLoc = maxLoc; }

 rectangle( img_display, matchLoc, Point(
    matchLoc.x + templ.cols , matchLoc.y +
    templ.rows ), Scalar::all(0), 2, 8, 0 );
 rectangle( result, matchLoc, Point( matchLoc.x
    + templ.cols , matchLoc.y + templ.rows ),
    Scalar::all(0), 2, 8, 0 );
```

No código acima vemos o desenho de um retângulo centrado no ponto de interesse calculado como ”melhor”na imagem a procurar, o outro retângulo é desenhado numa imagem adicional criado pelo algoritmo de processamento centrada no mesmo local.

A.3 Resultados e principais conclusões

Nos seguintes testes foi utilizada como template a procurar a seguinte imagem:
Num primeiro teste foi utilizada uma imagem padrão com vários sinais de trânsito, da qual a imagem template foi tirada, por isso esperávamos bons resultados que foram atingidos, sendo a imagem encontrada facilmente no sítio correto.

Figura 1: Imagem template

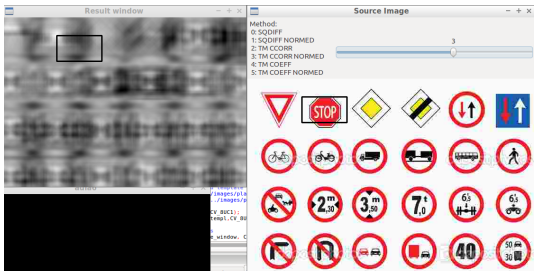

Figura 2: Resultado obtido após exercício 1

Após este teste decidimos ir buscar uma imagem de um ambiente mais real e com vários sinais de ”STOP”para ver como o programa reagiria. Os resultados encontrados foram também bons e interessantes, no sentido de que, nuns algoritmos a imagem template foi encontrada no sinal do meio(Figura 3), e com o algoritmo *TM COEFF NORMED(5)* foi encontrado o melhor ponto de ”match”no sinal da esquerda(Figura 4) como podemos ver nas imagens abaixo:

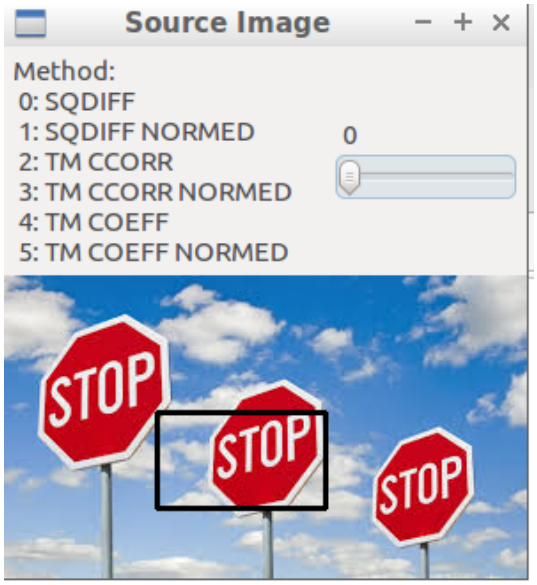

Figura 3: Resultado obtido após imagem 2

Figura 4: Resultado obtido após imagem 2 com outro algoritmo

Após estes testes ficamos intrigados e fomos ainda fazer um último teste rodando a imagem inicial 90º para a esquerda para ver como o template match reagiria, sendo a imagem inicial bem encontrada em 4 dos 5 algoritmos utilizados.

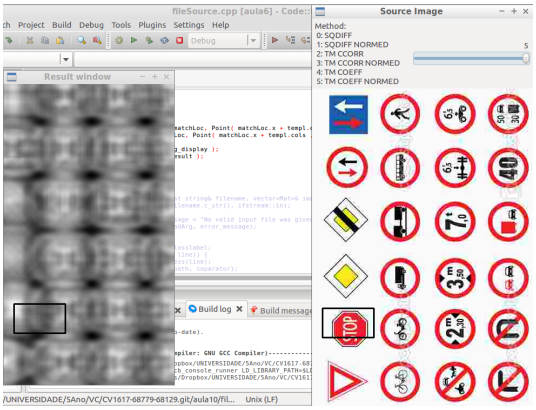

Figura 5: Resultado obtido após virar a Figura 1 90º

B. Problema #2

B.1 Enunciado

Develop a face detector and a face recognition program. You should consider the existence of a database with images containing the reference faces and your program has to determine the one that is most similar to the one that is under analysis. Explore the OpenCv tutorial about the use of cascade classifiers, as well as the other tutorials regarding face detection and face recognition

B.2 Resolução e principais conclusões

Para a resolução deste problema foi utilizado o tutorial *Cascade Classifier* disponível no site do opencv em `http://docs.opencv.org/2.4/doc/tutorials/objdetect/cascade_classifier/cascade_classifier.html`.

Para a elaboração deste problema foram realizadas as seguintes operações:

- Carregar ficheiros XML cascades (diposnivel nos repositorios do OpenCV): `haarcascade_frontalface_alt.xml` e `haarcascade_eye_tree_eyeglasses.xml`.
- Ler vídeo de stream `CvCapture`
- Aplicar o classificador a cada frame através do método `detectAndDisplay(Mat)`
  - É utilizado o seguinte método para detetar faces:
    ```
    face_cascade.detectMultiScale(
    frame_gray, faces, 1.1, 2,
    0|CV_HAAR_SCALE_IMAGE, Size(30,
    30) );
    ```
  - Para deteção dos olhos é usado o seguinte método
    ```
    eyes_cascade.detectMultiScale(
    faceROI, eyes, 1.1, 2, 0
    |CV_HAAR_SCALE_IMAGE, Size(30, 30)
    );
    ```

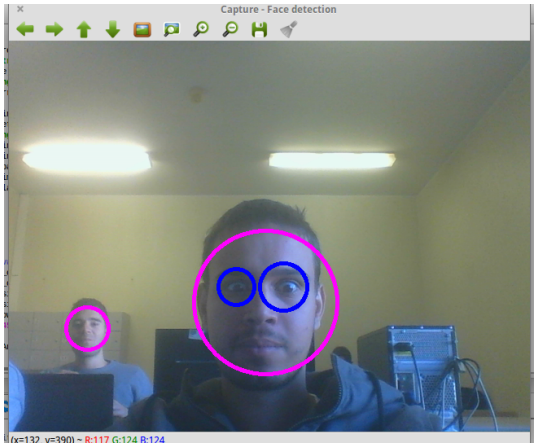

Figura 6: Resultado obtido após exercício 2

C. Exercícios adicionais/opcionais

Na questão do reconhecimento de sinais de trânsito pareceu-nos que o template matching não era o indicado para resolver o problema. Não só porque em imensos casos os algoritmos não encontravam o local certo na imagem a procurar, mas também porque esta implementação não ia funcionar em imensas circustâncias, como por exemplo, o ocultamento parcial do objeto a procurar.

Encontrámos um repositório com uma solução através da utilização de *Gielis curves* e seguindo os passos de instalação do README que pode ser encontrado aqui *https://github.com/glemaitre/traffic-sign-detection* conseguimos obter resultados surpreendentes.

O resultado para a primeira imagem utilizada por nós no exercício 1 é o seguinte:

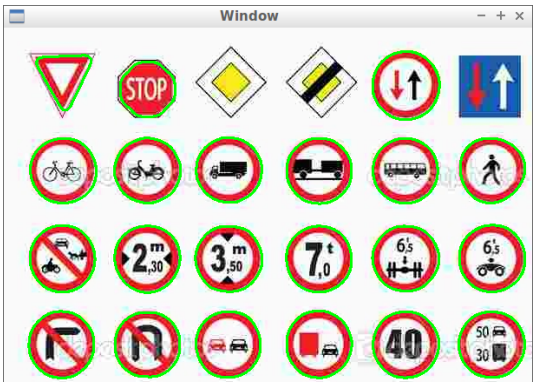

Figura 7: Resultado obtido após implementação adicional

O algoritmo usado aqui não usa template matching, mas é uma forma adicional muito interessante de resolver o exercício acima proposto. É primeiramente convertida a imagem rgb em ihls color space e em logarithmic chromatic vermelho e azul.

Listing 4: Image convertion

```
cv::Mat ihls_image;
colorconversion::convert_rgb_to_ihls(input_image,
    ihls_image);

std::vector< cv::Mat > log_image;
colorconversion::rgb_to_log_rb(input_image,
    log_image);
```

Depois é feito uma segmentação usando uma biblioteca específica

Listing 5: Biblioteca usada

```
#include <img_processing/segmentation.h>
segmentation::seg_norm_hue(ihls_image,
    nhs_image_seg_red, nhs_mode);
```

sendo a imagem processada similar a algo como:

E é com essa imagem que depois fazem merge na original para se extrair os contornos.

Listing 6: Reconstrução do contorno

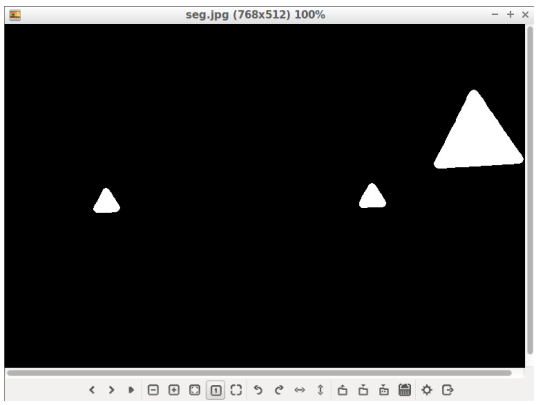

Figura 8: Resultado obtido após segmentação

Figura 9: Resultado obtido em imagem de contexto real

Figura 10: Outro resultado em imagem num contexto real

```
std::cout << "Contour #" << contour_idx << ":\n"
    << final_config << std::endl;
std::vector< cv::Point2f > gielis_contour;
int nb_points = 1000;
optimisation::gielis_reconstruction(final_config,
    gielis_contour, nb_points);

...

std::vector< cv::Point >
    distorted_gielis_contour_int
   (distorted_gielis_contour.size());
for (unsigned int i = 0;
   i < distorted_gielis_contour.size(); i++) {
   distorted_gielis_contour_int[i].x = (int)
       std::round(distorted_gielis_contour[i].x);
   distorted_gielis_contour_int[i].y = (int)
       std::round(distorted_gielis_contour[i].y);
}
```

Também é utlizado algoritmos para eliminar/corrigir a distorsão, normalizar o formato dos contornos, etc num código bem documentado. Este código pode ser encontrado na pasta traffic-sign-detection-master/src/main.cpp do nosso repositório dentro do diretório /aula10 e pode ser corrido da seguinte forma *bin/main ¡path para a imagem¿, ex: bin/main test-images/stop.jpg* que produzirá, por exemplo, os seguintes resultados:

REFERÊNCIAS

[1] Neves, A. J. R.; Dias, P. Slides teóricos Visão por Computador - Aula 10 (2016)

[2] OpenCV. Opencv Documentation. Web. 15 Outubro 2016.

[3] Github. traffic-sign-detection. 8 Setember 2015.