UNIVERSIDADE
LUSÓFONA

Relatório Projeto Final

Deteção e Reconhecimento de Jogos de Xadrez

André Jesus – a22207061

Tomás Nave – a22208623

www.ulusofona.pt

Índice

1. Introdução

1.1. Contexto e Objetivos
O xadrez é um jogo de tabuleiro amplamente reconhecido pela sua complexidade e profundidade estratégica. Este projeto visa desenvolver uma aplicação que identifique e reconheça tabuleiros de xadrez em imagens digitais, bem como as peças e as suas respetivas posições no tabuleiro. A aplicação deve ser capaz de lidar com variações em dimensões, iluminação e possíveis deformações nas imagens.

1.2. Descrição do Problema
O problema envolve a deteção do tabuleiro de xadrez numa imagem, a segmentação das peças e o reconhecimento da peça em cada posição do tabuleiro. A aplicação deve funcionar em diferentes níveis de complexidade, desde tabuleiros com fundo uniforme até imagens fotográficas com fundos irregulares e diferentes ângulos de visão.

1.3. Requisitos da Aplicação

A aplicação deve:
Oferecer uma interface para manipulação de imagens e aplicação das funções desenvolvidas.
Identificar a localização, dimensão e ângulo do tabuleiro.
Identificar a peça em cada casa do tabuleiro.
Lidar com variações de dimensão, iluminação e deformidade das imagens.

2. Metodologia

2.1. Pré-processamento de Imagens
O pré-processamento envolve a conversão de cores e binarização da imagem para facilitar a identificação do tabuleiro e das peças. Funções específicas foram implementadas para converter a imagem para o espaço de cores HSV e identificar componentes relevantes.

2.2. Deteção do Tabuleiro de Xadrez
Utilizámos métodos para detectar as margens do tabuleiro e segmentá-lo a partir da imagem. A técnica envolve a análise de histograma para determinar as áreas do tabuleiro que precisam de ser extraídas.

2.3. Segmentação das Peças
Cada peça é isolada ao cortar a imagem do tabuleiro em pequenas secções correspondentes a cada casa do tabuleiro. Estas secções são então binarizadas para facilitar a identificação das peças.

2.4. Reconhecimento das Peças
A identificação das peças é realizada comparando cada secção da imagem com uma base de dados de imagens de peças. Funções foram implementadas para calcular a semelhança entre as imagens e identificar a peça correspondente.

2.5. Classificação e Comparação de Imagens
A classificação das peças é feita através da comparação de cada secção da imagem com imagens de referência. Funções específicas foram implementadas para comparar a similaridade de pixels e identificar a peça.

3. Descrição das Funções Implementadas

3.1. ColorToHSV
Esta função converte uma cor do espaço de cores RGB para o espaço de cores HSV. Essa conversão é crucial para a segmentação baseada em cor, pois facilita a distinção entre diferentes cores numa imagem.

```
2 referências
public static void ColorToHSV(Color color, out double hue, out double saturation, out double value)
{
    int max = Math.Max(color.R, Math.Max(color.G, color.B));
    int min = Math.Min(color.R, Math.Min(color.G, color.B));

    hue = color.GetHue();
    saturation = (max == 0) ? 0 : 1d - (1d * min / max);
    value = max / 255d;

}
```

Descrição:

Essa função recebe uma cor no formato RGB e retorna seus componentes HSV (matiz, saturação e valor).

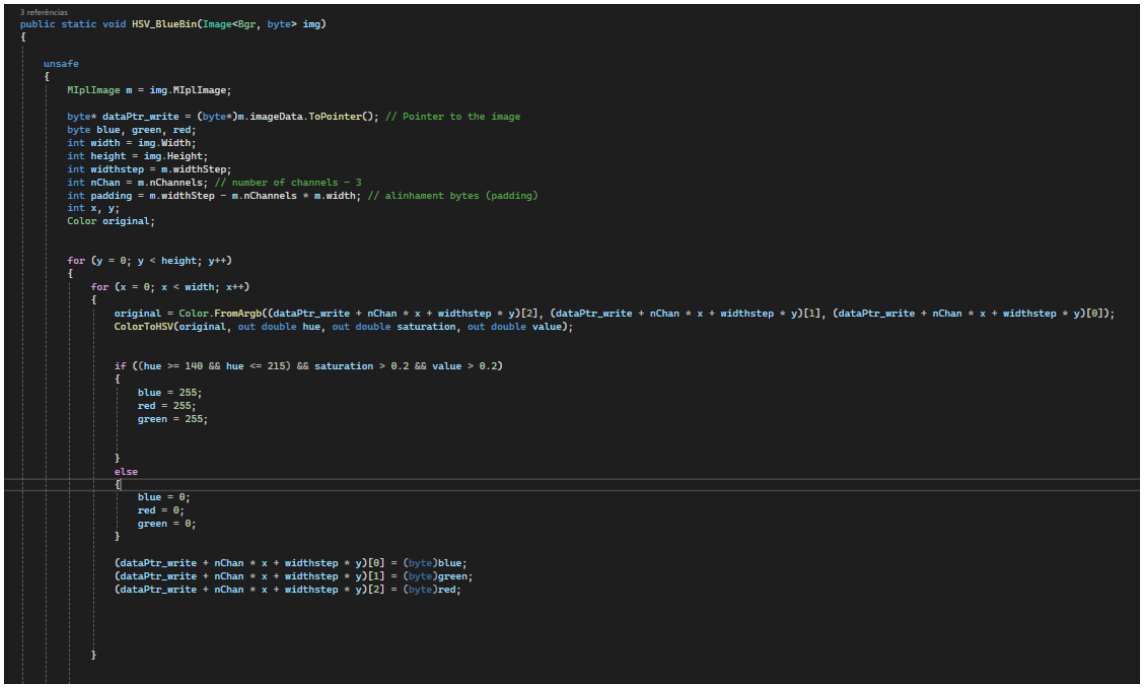
Primeiro, encontra os valores máximo e mínimo das componentes de cor RGB.

Em seguida, calcula a matriz usando o método GetHue () da estrutura Color.

A saturação é calculada com base no máximo e no mínimo das componentes de cor, enquanto o valor é simplesmente o máximo dividido por 255 (normalizado para um intervalo de 0 a 1).

3.2. HSV_BlueBin

Esta função aplica um filtro para identificar regiões azuis na imagem, convertendo essas regiões para branco e todas as outras regiões para preto. Este filtro é útil para isolar áreas do tabuleiro com uma determinada tonalidade de azul.



```
3 referências
public static void HSV_BlueBin(Image<Bgr, byte> img)
{

    unsafe
    {
        MIplImage m = img.MIplImage;

        byte* dataPtr_write = (byte*)m.imageData.ToPointer(); // Pointer to the image
        byte blue, green, red;
        int width = img.Width;
        int height = img.Height;
        int widthstep = m.widthStep;
        int nChan = m.nChannels; // number of channels - 3
        int padding = m.widthStep - m.nChannels * m.width; // alinhament bytes (padding)
        int x, y;
        Color original;


        for (y = 0; y < height; y++)
        {
            for (x = 0; x < width; x++)
            {
                original = Color.FromArgb((dataPtr_write + nChan * x + widthstep * y)[2], (dataPtr_write + nChan * x + widthstep * y)[1], (dataPtr_write + nChan * x + widthstep * y)[0]);
                ColorToHSV(original, out double hue, out double saturation, out double value);


                if ((hue >= 140 && hue <= 215) && saturation > 0.2 && value > 0.2)
                {
                    blue = 255;
                    red = 255;
                    green = 255;


                }
                else
                {
                    blue = 0;
                    red = 0;
                    green = 0;
                }

                (dataPtr_write + nChan * x + widthstep * y)[0] = (byte)blue;
                (dataPtr_write + nChan * x + widthstep * y)[1] = (byte)green;
                (dataPtr_write + nChan * x + widthstep * y)[2] = (byte)red;



            }
```

Descrição:

Nesta função vamos percorrer cada pixel da imagem e por cada pixel:

Extraímos os valores de Cor do Pixel

Passamos os valores para dentro da função ColorToHSV () onde esta função tem o objetivo de converte a cor extraída de BGR para o espaço de cor HSV, decompondo-a em matriz (hue), saturação (saturation) e valor (value).

Por fim verificamos se o pixel tem uma matriz (hue) dentro da faixa de azul e, se for o caso, binariza o pixel como branco, se não, binariza como preto.

3.3. cortarMargens

Esta função remove as margens desnecessárias ao redor do tabuleiro de xadrez, identificando as áreas relevantes com base na presença de componentes específicos da imagem.

```
public static Image<Bgr, byte> cortarMargens(Image<Bgr, byte> img, Image<Bgr, byte> imgor,double tr)
{

    unsafe
    {
        MIplImage m = img.MIplImage;

        byte* dataPtr = (byte*)m.imageData.ToPointer(); // Pointer to the image
        int width = img.Width;
        int height = img.Height;
        int widthstep = m.widthStep;
        int nChan = m.nChannels; // number of channels - 3
        int padding = m.widthStep - m.nChannels * m.width; // alinhament bytes (padding)
        int x, y;
        int[] max = new int[8];
        double[] hist_y = new double[height];
        double[] hist_x = new double[width];
        byte red, blue, green;

        System.Drawing.Rectangle recorta;

        for (y = 0; y < height; y++)
        {



            //imagem se contorno
            for (x = 0; x < width; x++)
            {

                red = (dataPtr + (nChan * (x)) + m.widthStep * (y))[0];
                green = (dataPtr + (nChan * (x)) + m.widthStep * (y))[1];
                blue = (dataPtr + (nChan * (x)) + m.widthStep * (y))[2];

                if(blue == 255 && green == 255 && red == 255)
                {
                    hist_y[y]++;

                }

            }


            hist_y[y] = 100.0 * hist_y[y] / width;

        }
    int index_yt = 0;
    for (y = 0; y < height; y++)
    {
        if (hist_y[y] > tr)
        {
            index_yt = y;
            break;

        }
    }

    int index_yb = height - 1;
    for (y = height - 1 ; y >= 0 ; y--)
    {
        if (hist_y[y] > tr)
        {
            index_yb = y;
            break;

        }
    }


    for (x = 0; x < width; x++)
    {



            //imagem se contorno
        for (y = 0; y < height; y++)
        {

            red = (dataPtr + (nChan * (x)) + m.widthStep * (y))[0];
            green = (dataPtr + (nChan * (x)) + m.widthStep * (y))[1];
            blue = (dataPtr + (nChan * (x)) + m.widthStep * (y))[2];

            if (blue == 255 && green == 255 && red == 255)
            {
                hist_x[x]++;

            }

        }


        hist_x[x] = 100.0 * hist_x[x] / height;

    }

    int index_xt = 0;
    for (x = 0; x < width; x++)
    {
        if (hist_x[x] > tr)
        {
            index_xt = x;
            break;

        }

    }

    int index_xb = width - 1;
    for (x = width - 1; x >= 0; x--)
    {
        if (hist_x[x] > tr)
        {
            index_xb = x;
            break;
        }
    }

    recorta = new System.Drawing.Rectangle(index_xt, index_yt, index_xb - index_xt, index_yb - index_yt);
    Image<Bgr, byte> imagem = imgor.Copy(recorta);
    return imagem;
```

Descrição:

O código percorre cada linha da imagem, pixel por pixel, e verifica se o pixel é branco (RGB = 255, 255, 255). Se for, incrementa o contador para o histograma vertical (hist_y). Depois, normaliza os valores do histograma para um intervalo de 0 a 100.

Depois determinamos os índices das margens superior e inferior com base nos valores normalizados do histograma. O limite de confiança threshold (tr) é usado para decidir se uma linha deve ser considerada como parte do objeto ou como parte do fundo.

Este threshold calcula o histograma horizontal, que é semelhante ao histograma vertical, mas percorre a imagem na direção horizontal.

Depois são determinados os índices das margens esquerda e direita, semelhantes à determinação das margens verticalmente.

Por fim, usando os índices das margens calculados anteriormente, uma região retangular de interesse é definida. Essa região é então recortada da imagem original imgor e retornada como resultado da função.

3.4. get_rectangle

Esta função retorna uma imagem correspondente a uma casa específica do tabuleiro de xadrez, baseada nos seus índices de linha e coluna.

```
1 referência
public static Image<Bgr, byte> get_rectangle(Image<Bgr, byte> imagem, int index_x, int index_y)
{
    unsafe
    {
        MIplImage m = imagem.MIplImage;

        byte* dataPtr_write = (byte*)m.imageData.ToPointer(); // Pointer to the image
        byte blue, green, red;
        int width = imagem.Width;
        int height = imagem.Height;
        int width_q = imagem.Width / 8;
        int height_q = imagem.Height / 8;

        int nChan = m.nChannels;

        System.Drawing.Rectangle recorta;

        int index_xt = (index_x - 1) * (imagem.Width / 8);
        int index_yt = (index_y - 1) * (imagem.Height / 8);

        recorta = new System.Drawing.Rectangle(index_xt, index_yt, imagem.Width / 8, imagem.Height / 8);
        Image<Bgr, byte> img = imagem.Copy(recorta);

        return img;
    }
}
```

Descrição:

Calcula as coordenadas do canto superior esquerdo do quadrante com base nos índices fornecidos (coordenadas do tabuleiro) e no tamanho da imagem.

Define um retângulo de recorte com base nas coordenadas calculadas e no tamanho de um oitavo da imagem, pois o tabuleiro é um 8x8. Então, recorta essa região da imagem original e a retorna como resultado.

3.5. binarizacaoPeca

Esta função binariza a imagem de uma peça de xadrez, destacando a peça num fundo preto e convertendo as áreas da peça em branco, se a peça for preta fica toda branca, mas no caso da peça ser preta passa os contornes a branco.

Peça Preta Binarizada

Peça Branca Binarizada

```
5 referências
public static void binarizacaoPeca(Image<Bgr, byte> img)
{

    unsafe
    {
        MIplImage m = img.MIplImage;

        byte* dataPtr_write = (byte*)m.imageData.ToPointer(); // Pointer to the image
        byte blue, green, red;
        int width = img.Width;
        int height = img.Height;
        int widthstep = m.widthStep;
        int nChan = m.nChannels; // number of channels - 3
        int padding = m.widthStep - m.nChannels * m.width; // alinhament bytes (padding)
        int x, y;
        Color original;


        for (y = 0; y < height; y++)
        {
            for (x = 0; x < width; x++)
            {
                original = Color.FromArgb((dataPtr_write + nChan * x + widthstep * y)[2], (dataPtr_write + nChan * x + widthstep * y)[1], (dataPtr_write + nChan * x + widthstep * y)[0]);
                ColorToHSV(original, out double hue, out double saturation, out double value);


                if ( value <= 0.3)
                {
                    blue = 255;
                    red = 255;
                    green = 255;


                }
                else
                {
                    blue = 0;
                    red = 0;
                    green = 0;
                }

                (dataPtr_write + nChan * x + widthstep * y)[0] = (byte)blue;
                (dataPtr_write + nChan * x + widthstep * y)[1] = (byte)green;
                (dataPtr_write + nChan * x + widthstep * y)[2] = (byte)red;




            }


        }
```

Descrição:

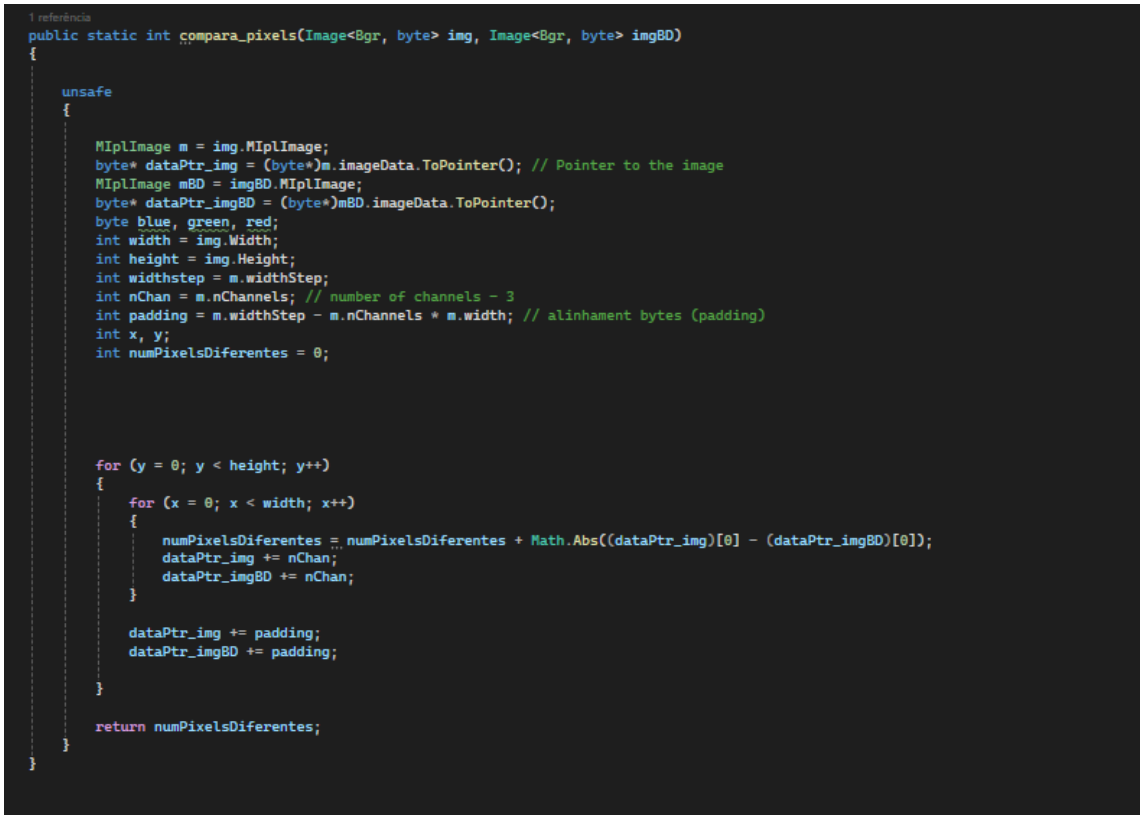
Passamos os valores para dentro da função ColorToHSV onde esta função tem o objetivo de converte a cor extraída de BGR para o espaço de cor HSV, decompondo-a em matriz (hue), saturação (saturation) e valor (value).

Itera sobre cada pixel da imagem e converte a sua cor para o espaço HSV utilizando a função ColorToHSV. Se o valor (brilho) do pixel for menor ou igual a 0.3, o pixel é definido como branco; caso contrário, é definido como preto.

Por fim define os valores de cor binarizados para os canais BGR do pixel, resultando na imagem binarizada.

3.6. compara_pixels

Compara cada pixel entre duas imagens e retorna uma pontuação de similaridade, se for muito alto significa que a imagem não é a mesma.



```
1 referência
public static int compara_pixels(Image<Bgr, byte> img, Image<Bgr, byte> imgBD)
{

    unsafe
    {

        MIplImage m = img.MIplImage;
        byte* dataPtr_img = (byte*)m.imageData.ToPointer(); // Pointer to the image
        MIplImage mBD = imgBD.MIplImage;
        byte* dataPtr_imgBD = (byte*)mBD.imageData.ToPointer();
        byte blue, green, red;
        int width = img.Width;
        int height = img.Height;
        int widthstep = m.widthStep;
        int nChan = m.nChannels; // number of channels - 3
        int padding = m.widthStep - m.nChannels * m.width; // alinhament bytes (padding)
        int x, y;
        int numPixelsDiferentes = 0;




        for (y = 0; y < height; y++)
        {
            for (x = 0; x < width; x++)
            {
                numPixelsDiferentes = numPixelsDiferentes + Math.Abs((dataPtr_img)[0] - (dataPtr_imgBD)[0]);
                dataPtr_img += nChan;
                dataPtr_imgBD += nChan;
            }

            dataPtr_img += padding;
            dataPtr_imgBD += padding;

        }

        return numPixelsDiferentes;
    }
}
```

Descrição:

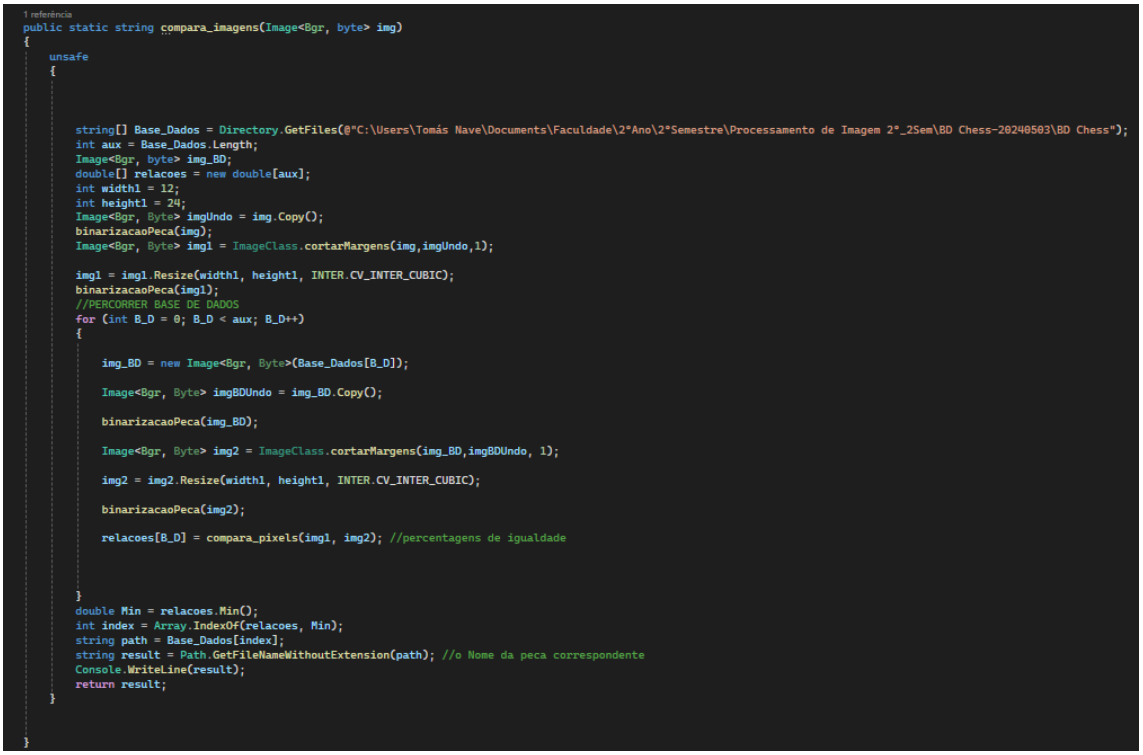
Esta função itera sobre cada pixel das duas imagens e compara os valores dos canais BGR de cada pixel.

Para cada pixel, calcula a diferença absoluta entre os valores de cor dos canais BGR.

Soma todas as diferenças calculadas para obter o número total de pixels diferentes entre as duas.

3.7. compara_imagens

Compara a imagem de uma peça com imagens de referência, presentes numa base de dados, para identificar a peça.



```
1 referência
public static string compara_imagens(Image<Bgr, byte> img)
{
    unsafe
    {



        string[] Base_Dados = Directory.GetFiles(@"C:\Users\Tomás Nave\Documents\Faculdade\2°Ano\2°Semestre\Processamento de Imagem 2°_2Sem\BD Chess-20240503\BD Chess");
        int aux = Base_Dados.Length;
        Image<Bgr, byte> img_BD;
        double[] relacoes = new double[aux];
        int width1 = 12;
        int height1 = 24;
        Image<Bgr, Byte> imgUndo = img.Copy();
        binarizacaoPeca(img);
        Image<Bgr, Byte> img1 = ImageClass.cortarMargens(img,imgUndo,1);

        img1 = img1.Resize(width1, height1, INTER.CV_INTER_CUBIC);
        binarizacaoPeca(img1);
        //PERCORRER BASE DE DADOS
        for (int B_D = 0; B_D < aux; B_D++)
        {

            img_BD = new Image<Bgr, Byte>(Base_Dados[B_D]);

            Image<Bgr, Byte> imgBDUndo = img_BD.Copy();

            binarizacaoPeca(img_BD);

            Image<Bgr, Byte> img2 = ImageClass.cortarMargens(img_BD,imgBDUndo, 1);

            img2 = img2.Resize(width1, height1, INTER.CV_INTER_CUBIC);

            binarizacaoPeca(img2);

            relacoes[B_D] = compara_pixels(img1, img2); //percentagens de igualdade



        }
        double Min = relacoes.Min();
        int index = Array.IndexOf(relacoes, Min);
        string path = Base_Dados[index];
        string result = Path.GetFileNameWithoutExtension(path); //o Nome da peca correspondente
        Console.WriteLine(result);
        return result;
    }


}
```

Descrição:

Obtém os caminhos das imagens na base de dados especificada.

Aplica a binarização (binarizacaoPeca) e o corte das margens na imagem de entrada e utiliza a função cortarMargens para que o recorte fique o mais correto possível. Esses passos preparam a imagem para a comparação com as imagens da base de dados.

Itera sobre cada imagem na base de dados.

Aplica a binarização, o corte das margens na imagem da base de dados e faz o resize da mesma para que fiquem todas do mesmo tamanho. Isso garante que as duas imagens estejam num formato possível de comparar.

Compara a imagem de entrada com cada imagem da base de dados usando a função **compara_pixels**. Calcula a percentagem de igualdade entre elas e armazena num array.

Encontra a menor percentagem de pixels diferentes e retorna o nome da peça correspondente.

3.8. isEmpty

Verifica se a casa está vazia comparando contando o número de pixels brancos. Esta função é utilizada para evitar perdas de tempo na comparação de casas vazias do tabuleiro.

```
1 referência
public static int isEmpty(Image<Bgr, byte> img)
{

    unsafe
    {
        MIplImage m = img.MIplImage;


        byte* dataPtrImg = (byte*)m.imageData.ToPointer();
        int nChan = m.nChannels;
        int padding = m.widthStep - m.nChannels * m.width;
        int x, y;
        int countPB = 0;


        for (y = 0; y < m.height; y++)
        {
            for (x = 0; x < m.width; x++)
            {
                if ((dataPtrImg + nChan * x + m.widthStep * y)[0] == 255)
                {
                    countPB++;

                }
            }
        }
        if (100*countPB / (m.width * m.height) < 2)
        {
            return 1;
        }
        else
        {
            return 0;
        }

    }
}
```

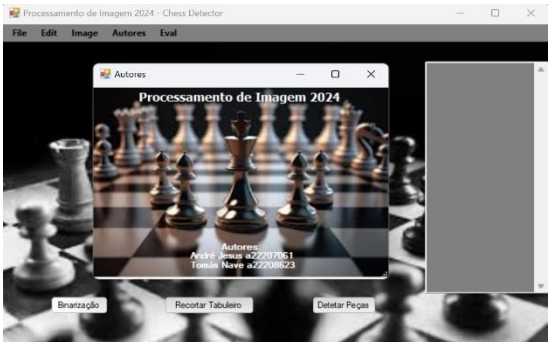
Descrição:

Recebe uma imagem que já foi binarizada anteriormente.

Itera sobre cada pixel da imagem e conta o número de pixels brancos.

Calcula a percentagem de pixels brancos em relação ao total de pixels na imagem, se for menor que 2%, considera-se que a imagem está vazia, caso contrário, considera-se que não está vazia.

4. Interface Gráfica



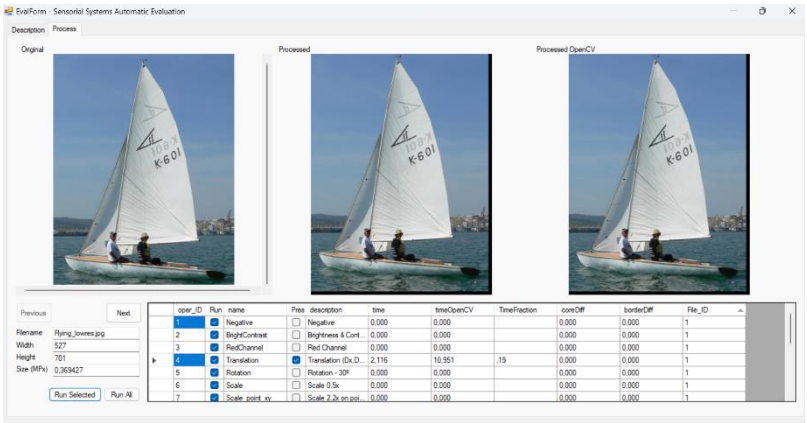
4.1. Explicação Interface Gráfica

Relativamente á interface gráfica, temos 3 botões, "Binarizacao", "Recortar Tabuleiro" e "Detetar Peca".
O botão "Binarizacao", como o próprio nome indica, binariza a imagem que terá sido selecionada anteriormente, o botão "Recortar Tabuleiro", recorta a imagem que terá sido selecionada anteriormente, deixando apenas o tabuleiro e recortando todas as margens,

Por fim o botão "Detetar Peças", deteta todas as peças presentes no tabuleiro comparando-as com as peças que estão presentes na base de dados, retornando uma lista que indica a posição de cada peça presente no tabuleiro. Sempre que se carrega num botão, para poder carregar noutro tem de se selecionar uma imagem outra vez, senão, não funciona.

Adicionamos também uma imagem de fundo, e personalizamos a parte gráfica que diz respeitos aos autores do projeto, colocando os nossos nomes bem como uma imagem de fundo.

Na barra de tarefas em cima temos também o botao eval , que nos leva a uma pagina diferente onde se tem acesso a todos os filtros que criamos ao longo do ano letivo.

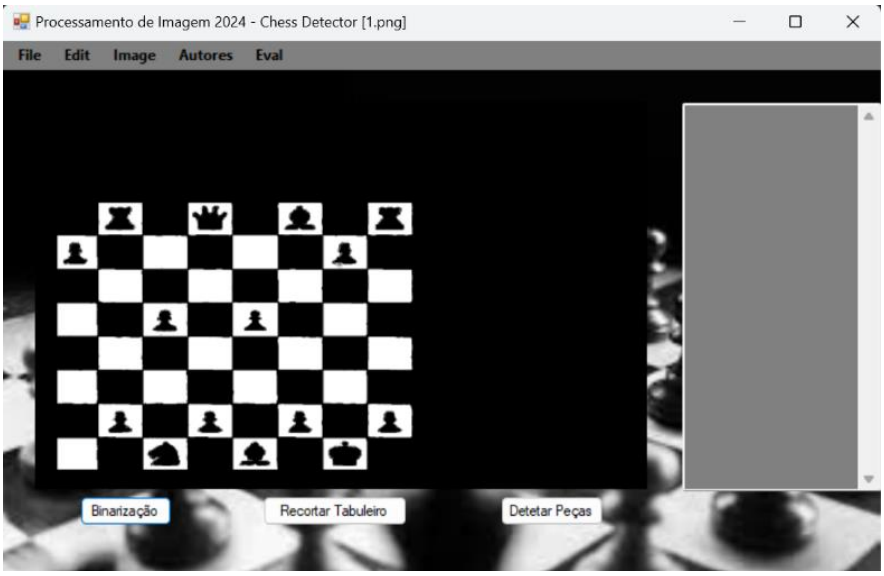


Aqui temos implementados todos os filtros obrigatórios implementados e os filtros Histogram_All e Histogram_RGB dos filtros complementares

4.2. Explicação individual das Funcionalidades da Interface

Aqui vamos explicar as varias funcionalidades do nosso programa de uma maneira mais detalhada

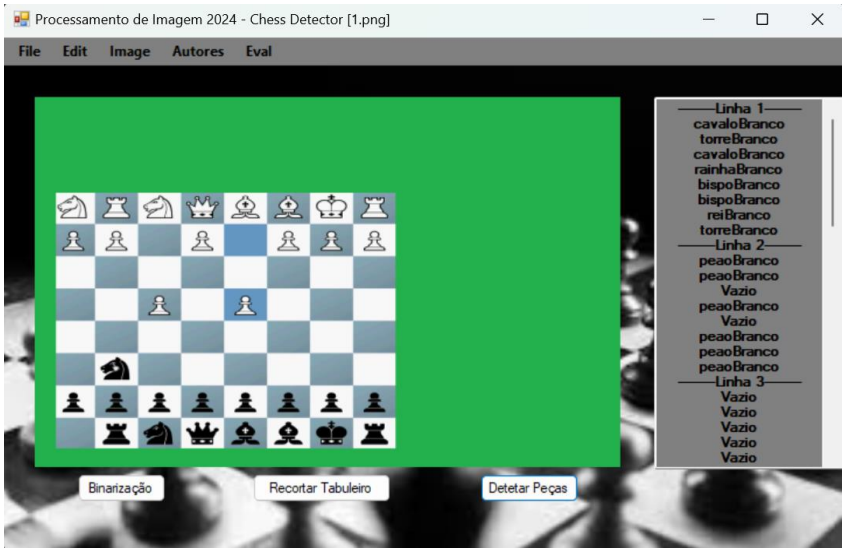
4.2.1 Botão Binarização




```
1 referência
private void button1_Click(object sender, EventArgs e)
{
    if (img == null) // verify if the image is already opened
        return;
    Cursor = Cursors.WaitCursor; // clock cursor

    imgUndo = img.Copy();

    ImageClass.HSV_BlueBin(img);



    ImageViewer.Image = img.Bitmap;
    ImageViewer.Refresh();

    Cursor = Cursors.Default; // normal cursor

}
```

O utlizador ao clicar neste botão, após escolher a imagem na qual quer aplicar transforações, esta função vai chamar a função HSV_BlueBin que já foi explicada anteriormente e vai retornar por fim a imagem binarizada.
Achamos esta implementação muito útil visto que é mais fácil de ver se estamos a fazer a binarização da imagem corretamente ou se é preciso fazer alguns ajustes

4.2.2 Botão Detetar Peças




```
1 referência
private void button2_Click(object sender, EventArgs e)
{
    if (img == null) // verify if the image is already opened
        return;
    Cursor = Cursors.WaitCursor; // clock cursor
    int width = img.Width;
    int height = img.Height;
    int g, j;
    //copy Undo Image
    imgUndo = img.Copy();

    ImageClass.HSV_BlueBin(img);

    Image<Bgr, byte> tabuleiroRecortado = ImageClass.cortarMargens(img, imgUndo, 1);

    textBox1.Text = "";

    for (g = 1; g < 9; g++)
    {
        textBox1.Text += "--------Linha " + g + "--------" + "\r\n";
        for (j = 1; j < 9; j++)
        {


            Image<Bgr, byte> pecaCortada = ImageClass.get_rectangle(tabuleiroRecortado, j, g);
            Image<Bgr, byte> imgCopy = pecaCortada.Copy();

            ImageViewer.Image = pecaCortada.Bitmap;
            ImageViewer.Refresh();

            ImageClass.binarizacaoPeca(imgCopy);


            int ee = ImageClass.isEmpty(imgCopy);
            if (ee == 1)
            {
                String path = "Vazio";
                textBox1.Text += path + "\r\n";
            }
            else
            {
                String path = ImageClass.compara_imagens(pecaCortada);
                textBox1.Text += path + "\r\n";
            }



        }

    }
    ImageViewer.Image = imgUndo.Bitmap;
    ImageViewer.Refresh();
    Cursor = Cursors.Default; // normal cursor
}
```

Este é o botão mais importante de todo o programa, este é o botão que aciona o código principal do programa e que vai juntar todas as funções explicadas anteriormente para chegar ao resultado esperado. Aqui o objetivo é todas as peças presentes no tabuleiro serem detetadas, compará-las com as peças que estão presentes na base de dados, retornando uma lista que indica a posição de cada peça presente no tabuleiro

Vamos agora fazer uma explicação mais detalhada de como funciona o programa do inicio ao fim.

Primeiro o programa recebe a imagem que contem o tabuleiro e faz a binarização da mesma chamando a função **HSV_BlueBin** que já foi explicada anteriormente (Secção 3.2. do relatório).

Logo em seguida o programa vai pegar na imagem já binarizada e vai passa-la como parâmetro para a função **cortarMargens** descrita anteriormente (Secção 3.3. do relatório) esta função vai cortar a imagem deixando apenas a parte relativa ao tabuleiro de xadres.

Depois o programa vai iterar por cada casa do tabuleiro e por cada casa vai fazer estes passos :

-Chama a função **get_rectangle** descrita anteriormente (Secção 3.4. do relatório), que obtém a imagem da peça localizada na posição específica do tabuleiro.

- Depois utiliza a imagem da peca obtida anteriormente e passa a como parâmetro na função **binarizacaoPeca** descrita anteriormente (Secção 3.5. do relatório) e esta função binariza a imagem da peça

-Depois é utilizada a função **isEmpty** descrita anteriormente (Secção 3.8. do relatório) para verificar se a imagem da peca que obtemos está vazia.

Se estiver vazia :

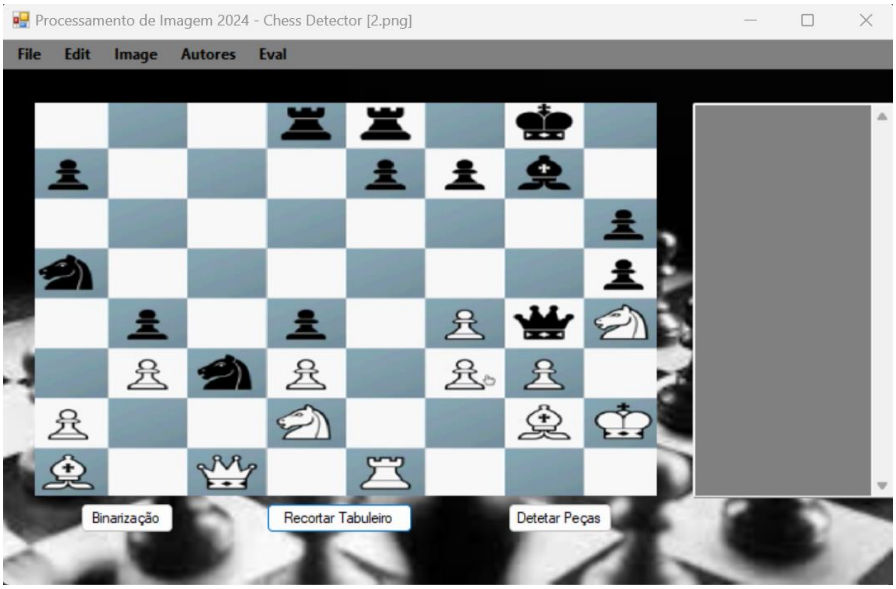
> É escrito na textBox da nossa interface “Vazio” indicando que esta casa esta vazia não tem nenhuma peca

Se não estiver vazia:

É chamada a função **compara_imagens** descrita anteriormente (Secção 3.8. do relatório) que vai comparar a imagem da peca com as imagens da base de dados retornando assim o nome da peca da base de dados que mais tem parecenças com a peca que esta na casa onde o programa se encontra

Este nome depois é escrito na textBox da nossa interface.

4.2.3 Botão Recortar Tabuleiro




```
1 referência
private void button3_Click(object sender, EventArgs e)
{

    if (img == null) // verify if the image is already opened
        return;
    Cursor = Cursors.WaitCursor; // clock cursor
    int width = img.Width;
    int height = img.Height;
    int g, j;
    //copy Undo Image
    imgUndo = img.Copy();

    ImageClass.HSV_BlueBin(img);

    Image<Bgr, byte> tabuleiroRecortado = ImageClass.cortarMargens(img, imgUndo, 1);

    ImageViewer.Image = tabuleiroRecortado.Bitmap;
    ImageViewer.Refresh();
    Cursor = Cursors.Default; // normal cursor

}
```

O utlizador ao clicar neste botão, após escolher a imagem na qual quer aplicar transforações, esta função vai chamar a função **HSV_BlueBin** que já foi explicada anteriormente e vai retornar a imagem binarizada.
Depois é chamada a função **cortarMargem** que vai recortar a imagem deixando apenas a parte relativa ao tabuleiro

Por fim a imagem já recortada é mostrada na interface
Achamos esta implementação muito útil visto que é mais fácil de ver se estamos a fazer o recorte do tabuleiro corretamente ou se é preciso fazer alguns ajustes

5. Resultados

5.1. Análise de Desempenho

Este trabalho poderia ser desenvolvido em 3 níveis:

• Nível 1 – Tabuleiro digital, em fundo uniforme, com um ângulo, entre o canto inferior esquerdo e o canto superior direito, de 45º.

A implementação que desenvolvemos consegue detetar todas as imagens de nível 1 de maneira correta

• Nível 2 – Tabuleiro digital com moldura simples, em fundo uniforme, com ângulo que pode ser diferente de 45º.

A implementação que desenvolvemos consegue detetar todas as imagens de nível 2 de maneira correta com exceção de todas as imagens que precisam do filtro de rotação, ou seja todas em com ângulo que pode ser diferente de 45º.
Tentamos fazer a implementação do mesmo mas tivemos alguns problemas em conseguir desenvolver a logica para obter as extremidades do tabuleiro de maneira correta.

• Nível 3 – Com dois tipos de imagens: tabuleiro digital com ângulo de 45º em fundo não-uniforme; e imagem fotográfica de Tabuleiro em fundo irregular, com diferentes ângulos.

A implementação que desenvolvemos consegue detetar apenas algumas imagens.

6. Conclusão

6.1. Conclusão Final

O desenvolvimento deste projeto proporcionou uma compreensão profunda de técnicas de processamento de imagem e sua aplicação prática em um problema específico: a identificação e classificação de peças em um tabuleiro de xadrez. O projeto envolveu uma série de desafios, desde a captura e processamento inicial das imagens até a identificação precisa das peças, exigindo uma combinação de conhecimento teórico e habilidades práticas em programação e manipulação de imagens.

Para nós o principal desafio foi a tentativa de desenvolver o filtro de rotação que não conseguimos concretizar

7.Processamento de imagem com LP1

Contactamos os alunos de LP1 por email no dia 18 de Maio, a pedir para nos adicionarem no discord para podermos agendar uma reunião.

Obtivemos resposta por parte dos alunos de LP1 dia 20 de Maio a dizer que não nos conseguiam adicionar no discord.

Só conseguimos ter contacto pelo discord com os alunos no dia 22 de Maio , onde marcamos a reunião para dia 24 de Maio.

No dia 24 de Maio fizemos uma reunião que teve uma duração de 40 minutos onde explicamos os filtros necessários para a implementação do trabalho dos alunos.
Disponibilizámos nos para responder a qualquer questão , e até em caso necessário fazer umas reuniões para explicar o necessário.

Acabamos por não obter mais nenhum contacto por parte dos alunos até a data da entrega, e ficamos a não saber se a nossa explicação foi a necessária.

No inicio da reunião perguntamos aos alunos se poderíamos gravar a aula, ao qual obtemos confirmação.

Deixamos em anexo no zip da entrega final do trabalho o vídeo da nossa reunião com os alunos.