Instituto de Matemática e Estatística
Universidade de São Paulo

EP1 - AMC Tests

Suayder Milhomem Costa - 12086131

Resumo

Neste artigo são apresentado os métodos e resultados obtidos nos experimentos feitos para ler o código da página e o número USP de cada prova, em resumo: dois métodos foram testados, um utilizando a técnica de blob detector e outro pagando simplesmente os componentes conexos e filtrando para chegar ao objeto desejado. Os resultados obtidos mostraram que o algoritmo proposto apenas utilizando componentes conexos se saiu sutilmente melhor.

Métodos e Resultados

Nesta seção será apresentado de forma descritiva o procedimento de desenvolvimento/pensamento de todo o método e o resultado e consequência de cada método aplicado. A seguir o trabalho foi dividido em três desafios principais para melhor entendimento, pre-processamento, detecção de círculo, tratamento de rotação, encontrar o código da prova, leitura do número USP.

A Figura 1 representa o fluxo geral do código desenvolvido. Onde, o bloco de pré-processamento recebe a imagem de entrada e a saída são strings representando o código USP (se houver) e o código da prova. O trabalho foca em testar dois métodos para detecção de objetos circulares na imagem, um é através da detecção de blobs o segundo é através de componente conexos, ambos estão localizados no bloco vermelho do fluxograma da solução, os demais blocos marcados de azul são partes comuns entre ambos os códigos a fim de tornar justa a comparação entre os métodos.

Figura 1: Fluxograma geral da solução. A parte em vermelho ”detecção de marcação circular”é a parte não comum para teste dos dois métodos testados

Pre-processamento

Olhando para o contexto de desafio, processar provas escaneadas e para a natureza das provas este bloco tem por objetivo preparar a prova para detectar os círculos então foram feitas algumas transformações a fim de remover o máximo de ruídos e objetos não semelhantes às marcações circulares.

O pré-processamento começa com uma normalização dos dados. Ao olhar para os dados, em geral não há muita variação nas provas originais, então para tornar o processo mais rápido as imagens são reduzidas o tamanho pela metade, ratio de 0.5 considerando que a perca de informações com essa redução não afeta o resultado final. Os algoritmos aplicados nesta primeira parte de normalização seguem a sequência:

1. redimensionamento do tamanho da imagem pela metade

2. rgb para grayscale
3. normalização de histograma
4. binarização da imagem

A saída depois de fazer este procedimento de normalização é representado na Figura 2, vale comentar sobre o método de binarização utilizado: Um método simples de threshold nem sempre se sai bem dependendo do caso da imagem pois a luminosidade pode variar então atrapalha para determinados casos, como exemplificado na Figura 3, para resolver isto foi utilizada a técnica de threshold adaptativo pois ele permite olhar para o contexto local da imagem para gerar um limiar, o resultado desta mudança é ilustrado na Figura 4.

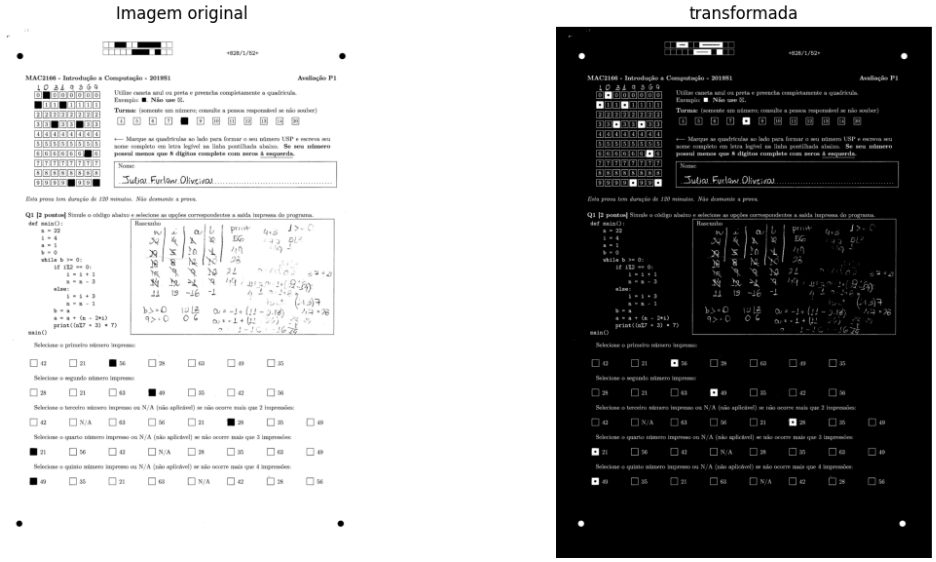

Figura 2: imagem original e normalizada.

Seguindo com o pre-processamento, a fim de remover ruídos na imagem e outros objetos foi aplicado uma transformação morfológica de *opening*, e no intuito de destacar somente as formas circulares o kernel para a transformação foi feito de forma que seja o mais circular possível e rodado em duas iterações para eliminar o máximo de objetos possível, incluindo o texto da prova, o resultado desta transformação pode ser visto na Figura 5.

Assim, com a imagem filtrada e os pontos de interesse destacados o último passo do pré-processamento é cortar a imagem para ter somente a área de interesse das provas, o cenário ideal é que a imagem seja cortada exatamente na borda dos círculos e (superiormente) no código da prova gerando uma imagem exata da área da prova. Nem sempre isto acontece.

Isso foi feito através da abordagem de contorno da imagem, isto é, todos os contornos da imagem foram pegos e então foi escolhido o de máximo contorno a fim de gerar um retângulo de mínima área que tem todo o contorno inscrito nele, em código é utilizada a função minAreaRect do OpenCV 4.5 que gerar o bounding box esperado e a respectiva rotação à qual o mesmo se encontra. A vantagem deste último método é que em alguns casos já é possível fazer uma transformação na imagem deixando ela totalmente na horizontal ou vertical assim facilita processos futuros relacionados a filtros ou transformações. A figura 6 ilustra o resultado deste processo.

A saída de todo este processo são duas imagens: 1) uma imagem com suas cores originais porém rotacionada e cortada como é a imagem ilustrada à direita na Figura 6, 2) uma imagem com as mesmas transformações da Figura

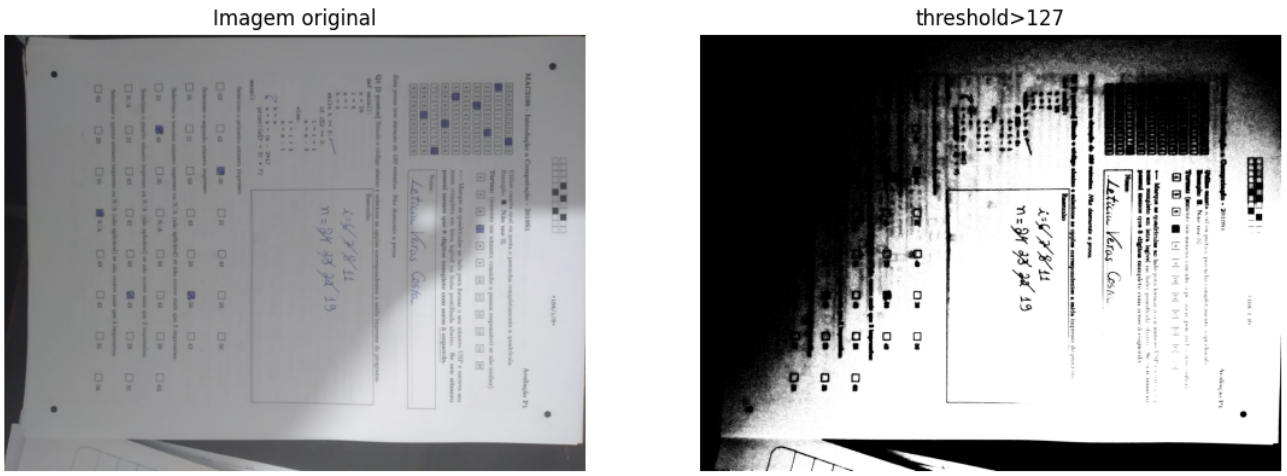

Figura 3: Imagem com simples threshold de 127 para tornar a imagem binária.

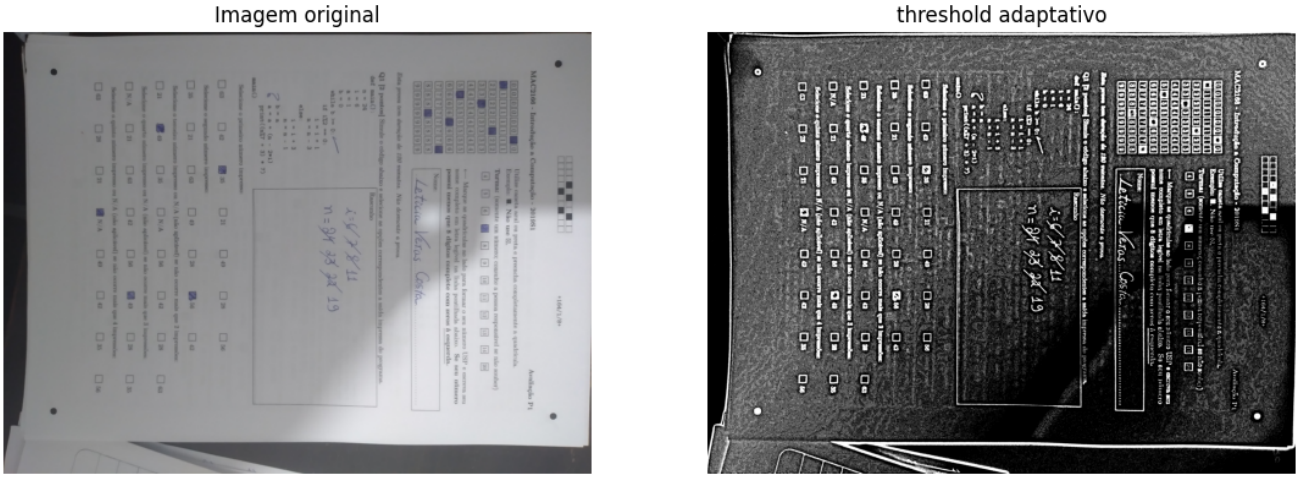

Figura 4: Imagem com threshold adaptativo

5 aplicada as mesmas transformações de rotação e crop da imagem anterior.

Detecção de marcação circular

Como mencionado, dois métodos de detecção de círculo foram usados ambos já implementados no OpenCV, são eles: SimpleBlobDetector e connectedComponentsWithStats. Em resumo os dois abordam componentes conexos, porém o primeiro tem uma implementação mais completa e complexa do que a abordagem somente por componentes conexos. A escola do segundo algoritmo foi na tentativa de a partir de uma maneira ainda mais simples resolver problemas que o outro não se sai bem, veremos pontos fortes e fracos e resultados de cada um.

Simple Blob Detector

Este método foi escolhido por ser um método relativamente simples mas que se utilizado corretamente é possível conseguir bons resultados.

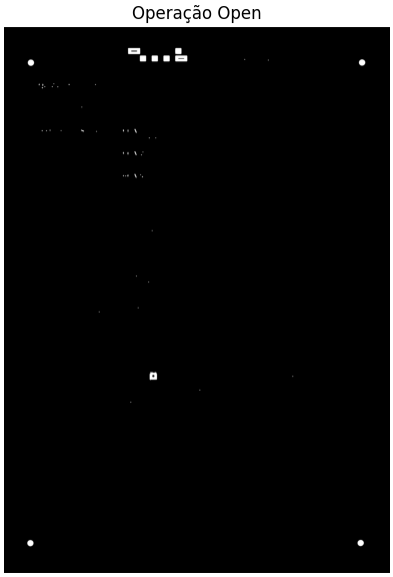

Figura 5: Operação Open na imagem removendo a maioria dos objetos e mantendo os círculos.

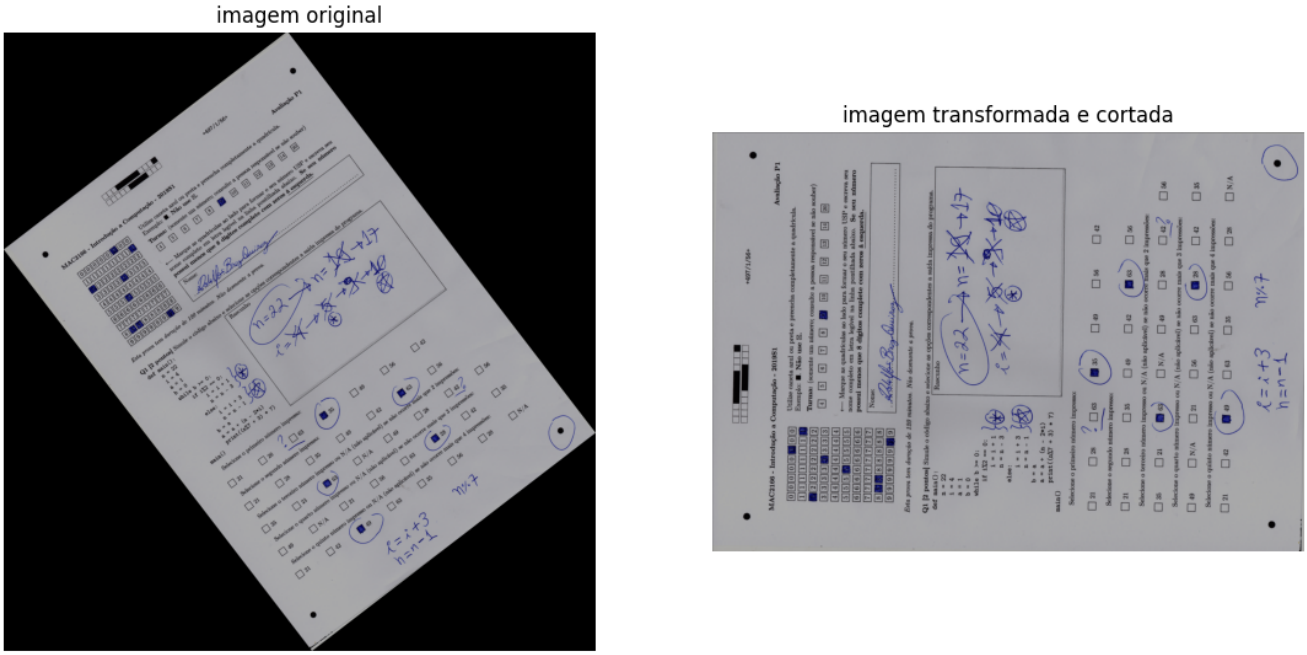

Figura 6: Imagem rotacionada e cortada

Sobre este método é interessante comentar sobre seus parâmetros, o sucesso ou falha dele se deve basicamente a escolha correta dos seus parâmetros de filtros. Para os testes realizados neste trabalho foi levado em consideração o filtro por Área, por cor e circularidade, isto é, convexidade.

O código abaixo ilustra os valores para cada parâmetro:

```
params = cv2.SimpleBlobDetector_Params()
params.filterByArea = True
params.minArea = int(np.pi*((rx_min*w/2.0)**2)) # area do circulo inscrito em um quadrado
params.maxArea = int(np.pi*((rx_max*w/2.0)**2))

```

```
params.filterByColor = True
params.blobColor=255

params.filterByCircularity = True
params.minCircularity = 0.01

detector = cv2.SimpleBlobDetector_create(params)
```

O filtro por cor e circularidade é mais simples, como a imagem já está binarizada temos apenas duas cores e queremos o que é branco (255) e a circularidade se relaciona a convexidade, quanto mais próximo de zero mais a imagem é convexa. No filtro por área é setado os valores de área mínima e máxima, estes valores dependem do tamanho da imagem, parâmetro w.

Para fazer este cálculo desta área, a fórmula principal é a da área de um círculo de raio r inscrito em um quadrado de tamanho $ratio_x \cdot w$. Isto é:

$$
\begin{aligned}
Area\_do\_circulo &= \pi \cdot r^2 \\
r &= \frac{ratio_x \cdot w}{2} \\
w &= \text{corresponde a largura da imagem} \\
ratio_x &= \text{relação entre o tamanho da imagem original e o diâmetro do círculo}
\end{aligned}
$$

Deste método o interessante a destacar é o $ratio\_x$ que quando multiplicado pelo largura da imagem retorna o tamanho estimado do diâmetro do círculo que estamos procurando.

Para obter este *ratio* basicamente foi feita uma operação em um pipeline paralelo ao qual pega uma porcentagem da base de dados na pasta ”originais”e tira um protótipo médio dos dados, isto é feito seguindo o mesmo protocolo de pre-processamento aplicado nas imagens, porém com a diferença que o redimensionamento gera imagens com as mesmas dimensões.

A Figura 7 representa o resultado deste processo de protótipo médio. E a partir dele podemos inferir algumas coisas quanto a imagem: marcado em vermelho, são os nossos objetos de interesse no momento, em verde é o código da prova que é algo que teremos interesse nos passos seguintes do pipeline. Ao visualizar estas marcações bem destacadas significa que a média é alta, ou seja, temos que a maioria dos pixels nas imagens se encontram nestas posições, é algo que reforça o quanto o pré-processamento foi eficaz em limpar vários objetos indesejados.

Com este protótipo médio em mãos foi feita uma inspeção manual à qual pegava valores aproximados para os tamanhos mínimos e máximos dos círculos e este valor dividido pelo tamanho da imagem gera os parâmetros rx_min e rx_max. O ponto fraco desta abordagem é que para imagens do dataset ”fotografadas”que tenham a perspectiva muito alterada o método pode não ser tão eficaz.

Ainda assim muitos pontos são retornados dependendo da imagem (ver Figura 8). Para resolver parte do excesso de pontos é aplicado um filtro simples que se resume em criar uma elipse inscrita no retângulo de modo que maximize a área de cobertura da elipse no retângulo, feito isso os elementos que se encontram dentro da elipse são excluídos, o trecho de código que faz esta operação:

```
# drop keypoints centered in the image. All elements in the ellipse are dropped
y_c, x_c = (np.array(preprocessed.shape)/2).astype(int) # center point

ellipse_eq = lambda x,y: (((x-x_c)**2)/(x_c-1)**2)+(((y-y_c)**2)/(y_c-1)**2)
ellipse_coef = np.array([ellipse_eq(k.pt[0], k.pt[1]) for k in keypoints])

keypoints = keypoints[ellipse_coef>=1]
```

A figura 9 ilustra o resultado desta operação.

Figura 7: Protótipo médio das imagens

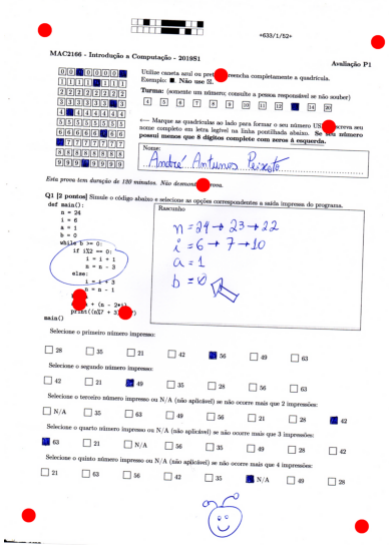

Figura 8: Pontos detectados ainda com os filtros aplicados no algoritmo

Por fim dois testes de sanidade dos dados são feitos, estes dois serão utilizados para verificação de resultados, basicamente representarão um tipo de erro do algoritmo. O primeiro teste consiste em verificar se restaram ao menos dois pontos, se isso não acontecer o programa retorna uma exceção e não continua com a prova. O segundo é consequência do primeiro, caso haja somente dois pontos na prova estes dois tem que fazer um ângulo cujo o cosseno deste arco seja próximo do valor do cosseno obtido pelo triangulo retângulo que pode ser gerado pelas dimensões da imagem, isto é, uma semelhança de cossenos que permite até determinado erro, caso esta condição seja violada novamente o código retorna uma exceção.

O retorno são os pontos restantes, que podem ser 2,3 ou 4, se houver mais que 4 é apenas escolhido os 4 mais extremos.

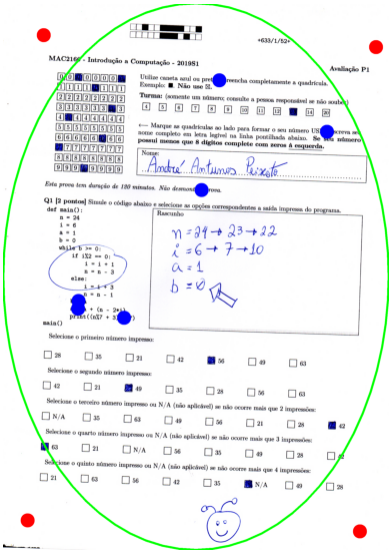

Figura 9: Os pontos destacados dentro da elipse foram excluídos e os pontos fora foram mantidos.

Componentes conexos

Este é um algoritmo que até certo ponto se assemelha bastante ao anterior. Porém alguns problemas de detecção de círculos foram notadas, por exemplo o problema ilustrado na Figura 10.

Assim, foi construída uma hipótese de que um algoritmo mais simples com os filtros corretos poderia gerar resultados possivelmente melhores e que contornam estes problemas de convexidade, o referido algoritmo é simplesmente um algoritmo de clusterização que pega os componentes conexos, no OpenCV: connectedComponentsWithStats.

É certo que o algoritmo inicialmente retornará muito mais ”keypoints”do que o algoritmo anterior porém aplicando os mesmos filtros é possível eliminar muitos pontos, os filtros aplicados foram:

- Filtro da elipse: mesmo filtro da subseção anterior: elementos dentro da elipse são excluídos elementos fora permanecem. A diferença aqui é na margem, ou seja, no filtro ainda é colocado uma margem para elementos muito próximos da elipse serem excluídos também.
- Filtro por área: o mesmo *max_area* e *min_area* utilizados no filtro de blob detector.
- Filtro por tamanho: em alguns casos o filtro por área não funciona, por exemplo o componente pode ser muito grande para um dos lados do mínimo retângulo que inscreve o objeto, estes casos são removidos com o filtro.
- Filtro pela relação entre a área coberta e o retângulo. Esta relação tem que chegar a um valor próximo de $\frac{\pi}{4}$.

Por fim as mesmas checagens feitas anteriormente foram aplicadas (tem menos que 2 pontos, se tiver dois pontos tem que fazer um ângulo próximo ao do triângulo-retângulo da imagem. E o retorno o mesmo também. A figura 11 ilustra o resultado final nas imagens

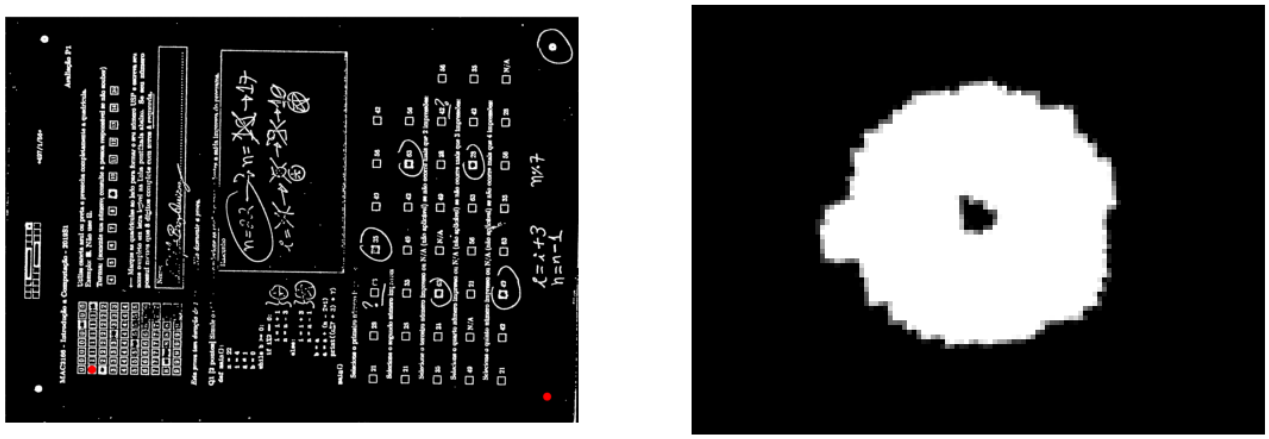

Figura 10: A imagem a esquerda contém pontos vermelhos que representam onde o algoritmo de blob detector detectou os círculos, à direita é um zoom do circulo *top-left* da mesma imagem, que mostra o motivo do algoritmo não ter considerado devido ao filtro de convexidade

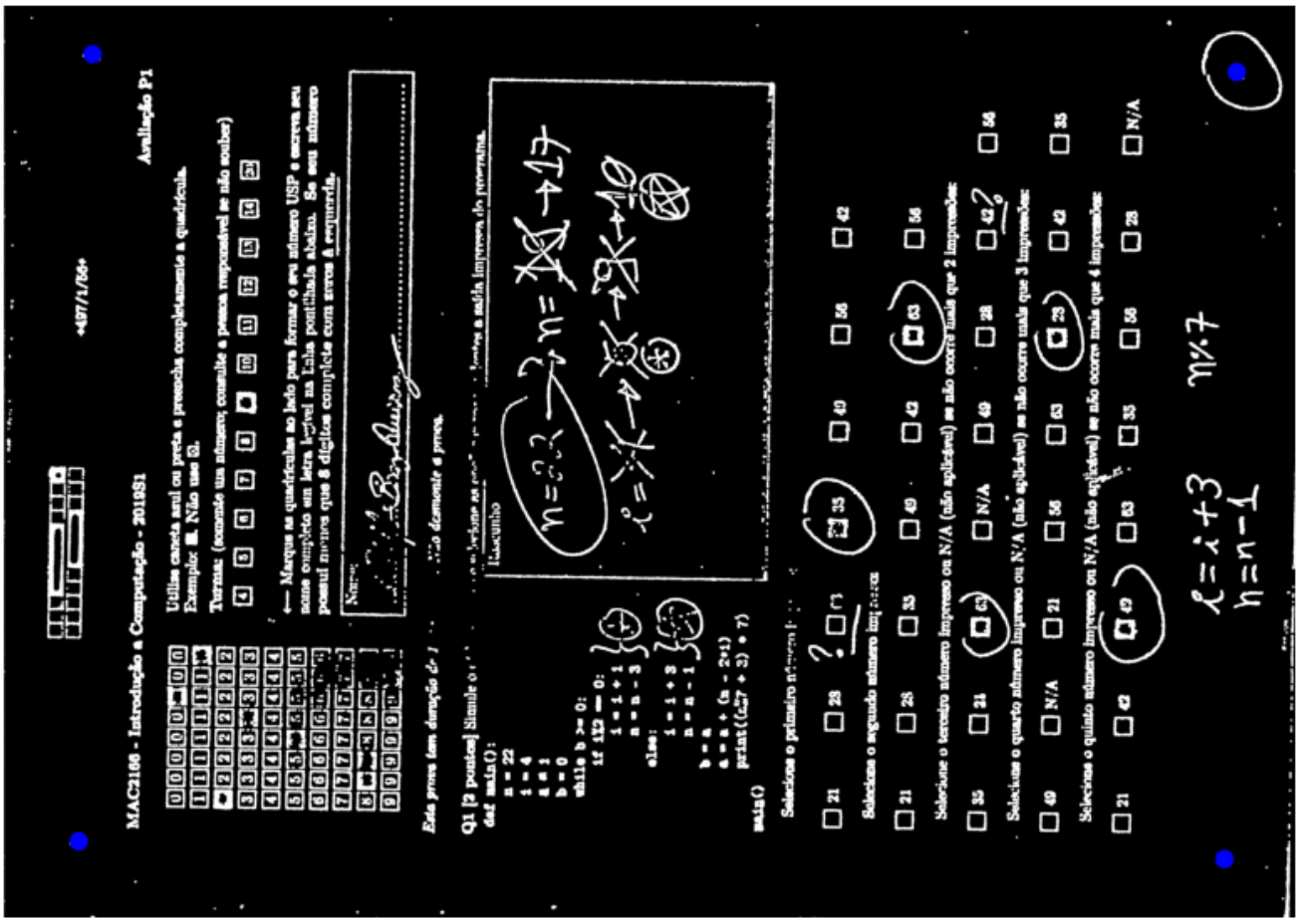

Figura 11: Imagem quando aplicado o código de componentes conexos e com os filtros aplicados

Tratamento de Rotação

Este bloco recebe como entrada a imagem[1] e os keypoints e a partir dos keypoints calcula a necessidade ou não de uma rotação e se necessário aplica, o bloco de código abaixo representa toda a função implementada:

[1] A imagem cortada e rotacionada na fase de pré-processamento

```python
def compute_rotation(keypoints, image):
    """
    Apply the rotation in the image and points and return these variables
    """

    points = np.array([k.pt for k in keypoints])
    points = points[points[:,0].argsort()]

    flag=0
    if isinstance(image, list):
        assert(len(image)==2)
        image, preprocessed = image
        flag=1

    h,w = image.shape[:2]
    centers = (w//2,h//2)

    idx = ((((points[1:,0]-points[0,0])**2)+((points[1:,1]-points[0,1])**2))**0.5).argmin()
    +1
    x = abs(points[idx, 1] - points[0,1])
    y = abs(points[idx, 0] - points[0,0])

    angle = math.atan2(x,y)
    angle = int(math.degrees(angle))
    angle = 0 if abs(angle-90)<10 else angle
    tx = 0
    ty = 0
    new_shape = (image.shape[1],image.shape[0])
    if centers[0]>centers[1]:
        angle = 90-angle
        tx = centers[1]-centers[0]
        ty = centers[0]-centers[1]
        new_shape = (new_shape[1], new_shape[0])

    rot_matrix = cv2.getRotationMatrix2D(centers,angle,1.0)
    rot_matrix[0,2]+= tx
    rot_matrix[1,2]+= ty

    image = cv2.warpAffine(image,rot_matrix, new_shape)

    if flag:
        preprocessed = cv2.warpAffine(preprocessed,rot_matrix, new_shape)
        return image, preprocessed, points

    points = cv2.transform(np.array([points]), rot_matrix).squeeze()

    return image, sort_kpts(points)
```

A lógica deste código: Ele considera que a imagem já está na horizontal ou na vertical, então a rotação será de um angulo α ou $90 - \alpha$, o 90 é usado para subtrair caso a imagem esteja na horizontal, o ângulo α é calculado a partir de dois ângulos próximos segundo a métrica euclidiana, pontos que estejam quase paralelos horizontalmente ou verticalmente. A transformação é aplicada tanto nos keypoints quanto na imagem.

O retorno desta função é a imagem com a rotação corrigida e os keypoints também rotacionados.

Encontrar código da prova

Para encontrar o código da prova há três desafios principais, são eles: encontrar o código, validar caso a área escolhida seja um código ou não e saber se a imagem está invertida, caso esteja uma rotação de 180 graus deve ser aplicada.

Para resolver estes problemas o problema de encontrar o código já é sabido os valores aproximados de onde estão os pontos do código portanto foi tratado somente o fato de caso não esteja disponível o ponto do círculo *top-left* ou *top-right* o faltante deve ser gerado a partir das coordenadas dos dois ou três pontos disponíveis. Assim é aplicado o cálculo normal da distância entre estes dois pontos e deles para o código:

```
# if the true top-left point is missing
if len(keypoints) <=3 and (top_left[0]>w/2 or top_left[1]>h/2):
    x = keypoints[:,0].min()
    y = keypoints[:,1].min()
    keypoints = np.append(keypoints,[[x, y]], axis=0)
    top_left = (int(x), int(y))
    top_right = (int(keypoints[0,0]), int(keypoints[0,1]))
    keypoints = sort_kpts(keypoints)

# if the true top-right point is missing
if len(keypoints) <=3 and (top_right[0]<w/2 or top_right[1]>h/2):
    x = keypoints[:,0].max()
    y = keypoints[:,1].min()
    keypoints = np.append(keypoints,[[x, y]], axis=0)
    top_right = (int(x), int(y))
    keypoints = sort_kpts(keypoints)


W = abs(top_right[0] - top_left[0])

tl_code = (int(W*0.259)+top_left[0], int(W*(-0.0435))+top_left[1])
br_code = (int(W*0.474)+top_left[0], int(W*(-0.0032))+top_left[1])
```

Para validar se encontramos o código ou não é aplicado na imagem um filtro de gaussiano seguido de um threshold (otsu) em toda a imagem então a área de interesse é cortada para ser aplicado o algoritmo detector de borda de Canny, isto nos permite ter as possíveis bordas do código, com isso a verificação feita é uma relação entre a quantidade de pixel brancos com o tamanho da área de interesse, se o erro passar de determinada margem a imagem será rotacionada e todo o processo é feito novamente, se nesta segunda faze for disparado o mesmo erro é porque não encontrou o código e o algoritmo para com a imagem corrente.

O motivo de aplicar um filtro passa baixa e uma limiarização em toda a imagem antes de cortar é pelo fato que pode gerar muito erro neste processo de limiarização que atrapalha os passos futuros.

A Figura 12 ilustra o resultado deste processo.

O resultado deste bloco são os valores correspondentes ao número do teste, número da página e número de verificação, este último é utilizado para fazer um ”health check”dos códigos lidos, se não bater com o valor esperado é disparada uma exceção de erro.

Ler número USP

Por fim, o número USP só e verificado caso a imagem corresponda à pagina 1 da prova.

O método de verificação e leitura segue o mesmo fluxo que o anterior porém desta vez pegando as coordenadas referentes ao número USP para ser lido e não trata caso a prova esteja rotacionada ou não pois isto já foi tratado anteriormente.

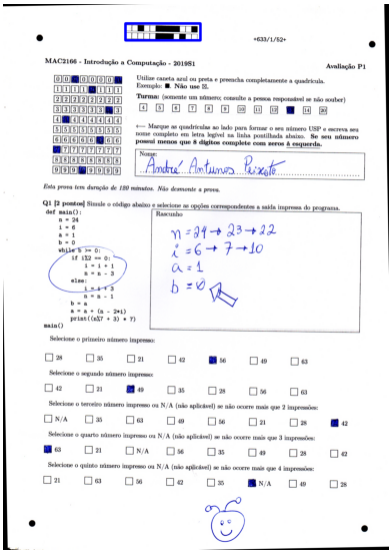

Figura 12: Com a marcação azul o código da prova encontrado. Nas bordas da imagem é possível observar algumas manchas pretas que foram geradas ao corrigir a rotação da imagem no passo de correção de rotação.

Resultados

Os testes foram rodados em todas as imagens da base de dados, imagens originais e fotografadas, os resultados são detalhados conforme o tipo de erro apresentado durante cada fase do algoritmo. Três tipos de erro foram mapeados:

1. **InsuficientPointsError** - este erro é disparado quando não há a quantidade necessária de pontos (círculos) para determinar e extrair os objetos e dados durante o restante do processo, este erro é disparado na fase de detecção de marcações circulares.

2. **InvertedImageException** - Esta exceção é disparada caso não seja encontrado nenhum código da prova na imagem. O cuidado com esta exceção é que para ser considerada errada deve ser disparada duas vezes, a primeira representa que a imagem será rotacionada para procurar o código, a segunda que realmente é contada como erro.

3. **DoesNotMatchCodeException** - quando o código da prova é extraído, também é lido o código de verificação, este código de verificação nos permite, através de uma fórmula matemática pré determinada, verificar se lemos os valores errados ou certos.

Na Tabela 1 há os valores dos resultados para os testes de performance do algoritmo de blob detector que nitidamente é possível ver a quantidade de erros gerados, nas imagens fotografadas principalmente que do total do dataset somente 14 foram lidas corretamente e 12 falsos positivos gerados (coluna DoesNotMatchCodeException), 14 erros por não conseguir detectar corretamente o ponto circular. Em resumo, para o dataset de imagens ”fotografadas”obtivemos uma acurácia de 33% e é bem ruim para o problema em questão.

Para o dataset ”fotografadas”o código teve uma performance não muito boa, para o dataset ”original”a performance já foi consideravelmente melhor como já esperado, não errando em nenhum passo de detecção de pontos ou do código da prova, errando apenas em algumas leituras.

O mesmo procedimento de teste foi feito utilizando o algoritmo connectedComponentsWithStats, os resultados também ilustrados na Tabela 2 mostram a mesma performance com relação às fotos Originais e uma sutil melhoria na base de dados mais caótica que é a base de dados de imagens fotografadas, que agora tem uma taxa de acerto de 42%.

Tabela 1: Resultado para o SimpleBlobDetector

dataset	Acertos	**InsuficientPointsError**	**InvertedImageException**	**DoesNotMatchCodeException**
Fotografadas	14	14	2	12
Originais	237	0	0	3

Tabela 2: Resultado do algoritmo connectedComponentsWithStats

dataset	Acertos	**InsuficientPointsError**	**InvertedImageException**	**DoesNotMatchCodeException**
Fotografadas	18	9	4	11
Originais	237	0	0	3

Ainda assim o algoritmo com componentes conexos conseguiu se sair um pouco melhor. O erro nas imagens originais foram investigados, o motivo da-se não por ler errado o código da prova e sim na hora de verifica-lo que onde o número de verificação é 60 o resultado na fórmula aplicada é 0 que é o resto da divisão por 60, este valor provavelmente deveria ser conferido tratado porém em uma verificação manual dos valores retornados foi confirmado o código ter lido corretamente. Não foi possível investigar o erro de cada para saber se há algum ponto de filtro nos algoritmos ou de preprocessamento nas imagens que possam melhorar os resultados, por exemplo, correções de perspectiva poderiam ajudar bastante porém é preciso ter uma melhor detecção dos pontos.

Código

O código pode ser encontrado neste repositório: https://github.com/suayder/lab_computer_vision