Laboratório de Visão Computacional – EP01

Nome: Lucas Martinuzzo Batista

O primeiro EP pede que sejam imagens de provas sejam analisadas para identificar três coisas: os marcadores circulares nos quatro cantos da prova, os códigos da prova (número da prova, da página e código de verificação), e, quando disponível, a matrícula do aluno.

SimpleBlobDetector

O primeiro método que eu usei para a primeira etapa, é a detecção de *blob* por limiarização.

Primeiro, utilizei a função *cv2.SimpleBlobDetector_create()* com os parâmetros padrões para ver o resultado e nenhum dos quatro marcadores foram capturados. Portanto, comecei a mexer nos parâmetros.

Primeiro eu tentei filtrar por cor, porém, mesmo filtrando só os *blobs* pretos, o algoritmo pega *blobs* brancos com centro preto, como os quadrados com números dentro das caixas de NUSP. Depois tentei filtrar por circularidade para eliminar os quadrados e focar só nos marcadores da borda.

Funcionou quase perfeitamente no primeiro arquivo de teste, **mac2166-t8.PDF-page-001-000.pbm**, depois de alguns ajustes (circularidade de 0.8 a 1), capturando as 4 bordas e um dos números do NUSP que resolvi filtrar por posição. Porém, ao testar arquivos com maiores distorções, não funcionou, então troquei novamente os parâmetros para capturar todos os *blobs* com cor 0, preto, e entre um intervalo de *minArea* e *maxArea* obtidos empiricamente. Este método funcionou nos scans_A, categoria de scans quase perfeitas, mas não nos scans_B, provas que foram editadas pelo professor para simular capturas mais diversas. Notei dois motivos para isso: a cor do preto nem sempre é zero e o tamanho da imagem é variável.

Eu solucionei o problema das cores primeiro normalizando, para ter certeza que a cor mais escura é 0 e a mais clara é 255 e depois aplicando um *threshold* de 200 para binarizar a prova. O problema da diferença de tamanhos foi solucionado ao redimensionar todas as imagens para um tamanho fixo.

Uma vez que as provas foram "normalizadas" com os tratamentos acima, o filtro de *blobs* por tamanho e cor funcionou, capturando os marcadores e mais alguns *blobs* no meio da prova. Filtrei apenas os da borda selecionando os 4 *blobs* mais externos.

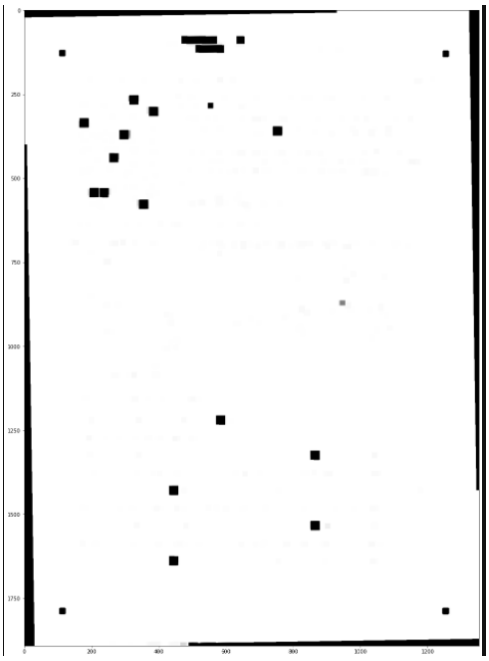
Algumas provas foram digitalizadas um pouco tortas, então utilizando a posição dos marcadores a esquerda, calculei o arco tangente para obter um ângulo de correção, utilizado para deixar as provas totalmente retas.

Como o próximo passo é pegar o código de da prova, comecei a testar parâmetros para captura-lo utilizando *blobs*. Porém notei que, por razões desconhecidas, não consegui fazer o algoritmo reconhecer as marcações pretas do código. Por isso, tentei fazer o inverso, pegar as marcações vazias e inverter o código gerado. Utilizando a cor branca e um intervalo de *minArea* e *maxArea* consegui capturar as áreas brancas nos códigos.

Para saber se as marcações estavam na parte de cima da prova, fiz a média das coordenadas dos pontos, assim caso algum ponto externo fosse capturado, sua influencia seria diluída, e verifiquei em qual extremidade estava esta coordenada média fazendo a correção de 90, 180 ou 270 graus, quando necessário.

Inicialmente, usei a posição dos *blobs* brancos para obter os códigos da prova através da diferença de suas posições ao início dos marcadores. Calculei a distância de cada *blob* em relação ao canto superior esquerdo do código, obtido através da fórmula no enunciado, e dividi pelo tamanho médio dos *blobs*, isso me dava posição relativa de quais códigos eram brancos.

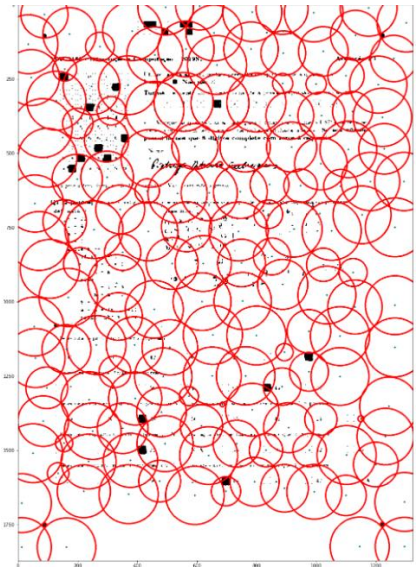
Infelizmente, em testes posteriores, nem todos os *blobs* brancos eram capturados ou eles tinham tamanhos um pouco distintos, o que causava que, por diferenças decimais, a fórmula resultava em posições distintas da correta. Por isso abandonei as coordenadas obtidas por *blobDetector*, e passei a utilizar somente informações das posições dos cantos dos retângulos dos códigos dados no enunciado. Calculei o tamanho médio de cada binário do código e verifiquei as cores de seus centros para determinar seu valor obtendo o número da prova, da página e o código verificador. Usei a fórmula de código verificador para garantir que os valores extraídos estavam corretos e passei para o último passo, a leitura do código USP.



Para a leitura dos códigos USP, fiz tratamentos morfológicos para fortificar as marcações do NUSP e eliminar as caixas e números restantes, pois estavam atrapalhando a captura. O resultado pode ser visto ao lado.

Através do tamanho médio destes quadrados e da posição fixa deles, extraí os blobs do NUSP e calculei suas posições em relação ao canto superior esquerdo para obter o número USP.

HoughTransform



O segundo método que usei foi o *cv2.HoughCircle()*, primeiro eu apliquei um *blur* na imagem, como recomendado para evitar detecção de falsos círculos. Posteriormente apliquei o filtro. Ele detectou muitos círculos indesejados, inclusive onde não havia algum, como pode ser visto ao lado. Por isso, fiz um ajuste fino dos parâmetros até obter o resultado desejado. Os principais parâmetros que ajustei foi o tamanho dos círculos, que coloquei entre um raio de 7 e 15 pixels, e o param2, um *threshold* que indica quão confiável o círculo é, portanto quanto menor o *threshold*, mais círculos aparecerão. O valor escolhido foi 10, resultado na detecção de apenas os marcadores. Como garantia, caso outras imagens capturem mais círculos (o que foi de fato observado em testes), eu mantive o filtro utilizado no primeiro método de selecionar apenas os 4 pontos mais externos.

Como foi pedido que apenas a etapa da identificação das marcações circulares dos cantos fosse realizada de duas maneiras diferentes, o resto do programa utilizou a mesma implementação da solução passada.

Considerações finais

Em geral os algoritmos funcionaram relativamente bem, exceto em casos que um X foi desenhado nos marcadores externos, inclusive na imagem amc016 onde o X é apenas uma sombra.