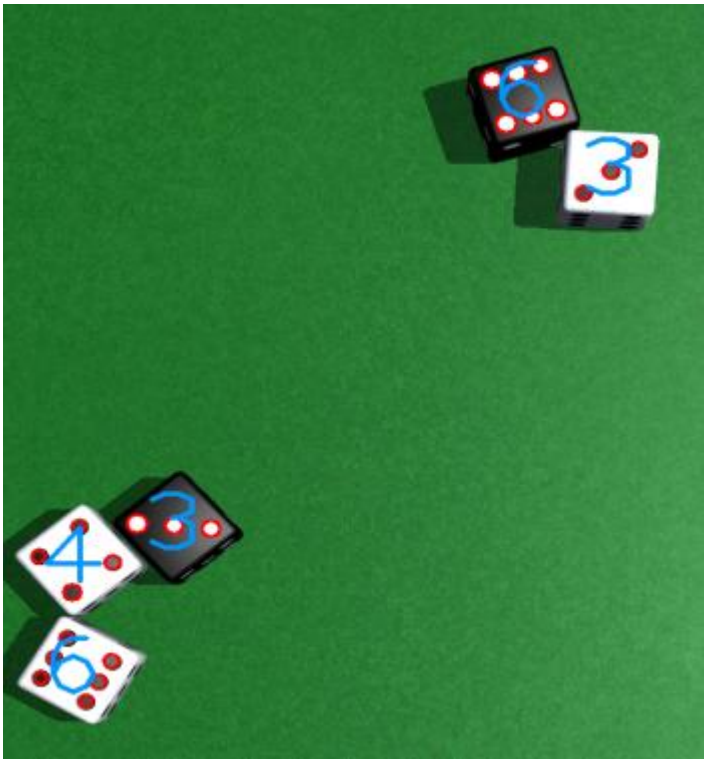


DOBÓKOCKA FELISMERÉS

ABSTRACT
Algoritmus dobokockák azonosítására és értékeik meghatározására, különféle körülmények között.

Molnár Bertalan, N2A06U

Gépi látás (GKLB_INTM038)

Contents

1. Dobókocka felismerés 1
2. Elmélet 2
   2.1. Adaptív küszöbölés 2
   2.2. SimpleBlobDetector 3
   2.3. DBscan 4
3. Részletes leírás 4
   3.1. Inicializálás 4
   3.2. Fő ciklus 6
   3.3. Képek feldolgozása 7
   3.4. Pöttyök felismerése 8
   3.5. Kockák csoportosítása 9
   3.6. Eredmény grafikus megjelenítése 13
4. Tesztelés 14
5. Fejlesztési lehetőségek 14
6. Felhasználói leírás 15
7. Források 16
8. Forráskód 17

1. Dobókocka felismerés

Dobókocka számláló alkalmazás: Különféle körülmények között dobokockák azonosítása és értékeik meghatározása.

https://github.com/MolnarBertalan/dicecounter

A program célja klasszikus 6 oldalú dobókockákról készült képen a pöttyök felismerése, és az egyes kockák elkülönítése, a kockákkal dobott értékek meghatározása.

Mivel alapvető elvárás a dobókockákkal szemben, hogy emberi szem által is könnyen olvashatóak legyenek. Ezért feltételezhetem, hogy a pöttyök és a kockatest színe jól elkülönül egymástól.

Ez alapján a pöttyök éle valószínűleg könnyen felismerhető.

A dobókockák azonosítását ezután a pöttyök csoportosításával lehet például elvégezni. Az egyes kockákon lévő pöttyök egymáshoz nagy valószínűséggel közelebb álnak, mint a többi pöttyhöz. Ezt clusterezési módszerekkel lehet megoldani.

Az alább ismertetett algoritmust a repositroyban megtalálható 109 képen teszteltem amelyek kölünböző színű, helyzetű, darabszámú kockákról készültek, eltérő hátterek előtt és különböző nagyításban.

A képeket egy androidos dobókockaszimulátor applikáció segítségével készítettem, ahol lehet változtatni a kockák színét, darabszámát, a háttér színét, és a nagyítást is.

Az eredményt csv fájlokban rögzítettem. Result_True a 100% ban pontos (természetesen emberi) eredményt tartalmazza összehasonlítás érdekében.

A feladat során több hibát okozott, hogy a clusterezés során egymáshoz közel álló kockák egy clusterbe kerültek, mivel a 2 es oldalon a pöttyök távolsága viszonylag nagy.

Ezt lépcsős clusterezéssel igyekeztem kiküszöbölni, ami 4 lépésben azonosítja a kockákat. Előbb a 6-os oldalakat, majd egyszerre a 3-as, 5-ös dobásokat, harmadjára a 4-est, és végül a 2-es, 1-es csoportokat.

Ezzel az egymáshoz közel álló 6,3,5 ös oldalak egyértelműen elkülöníthetőek, a 4-es, 2-es viszont még okozott nehézséget.

2. Elmélet

Az algoritmus a képeket előbb szürkeárnyaltosra alakítja, majd adaptív gauss küszöböléssel fekete-fehér képet állít elő. Ezen az 'előkezelt' képen OpenCV SimpleBlobdetection függvényével köröket keres, majd a körök középpontjait sklearn DBscan algoritmusával csoportosítja 'kockákká'.

2.1. Adaptív küszöbölés

Az adaptív küszöbölés a képek csak egy lokális részét dolgozza fel, és ezekre a részekre dinamikusan határozza meg a küszöb értéket. Ezért a vizsgált részeken belüli színváltozásokat emeli ki.

A módszer megfelelő működéséhez elengehetetlen, hogy a vizsgált képrészlet megfelelően nagy legyen, és egyszerre tartalmazza a kép lényegi részeit és hátteret is, így azok elkülöníthetőek.

cv.adaptiveThreshold (src, maxValue, adaptiveMethod, thresholdType, blockSize, C)

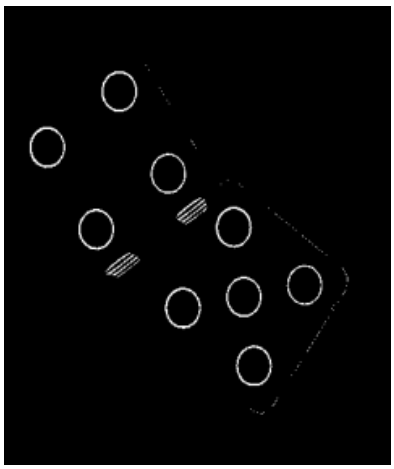
Ahol:

src	a feldolgozandó kép,
maxValue	a megengedett maximum intenzitás,
adaptiveMethod	a küszöb meghatározásának módszere,
thresholdType	a kimeneti intenzitás függvény,
blockSize	a mátrix mérete,
C	egy konstans ami a küszöb finomhangolására használható (T'=T-C)

AdaptiveMethod-nak Gauss módszert választottam, ami súlyozott átlagolást végez, és a vizsgált blokk közepén lévő pixeleket nagyobb súllyal látja el.

Jelen esetben kis mátrix esetén is a pöttyök körvonala kiemelhető. A mátrix méretének alsó korlátjának kiválasztásakor lényeges, hogy ez a körvonal egybefüggő legyen (blockSize). Természetesen kisebb BlockSize kisebb számolásigénnyel jár. A tesztek alapján blockSize = 9 (9*9 es mátrix) megfelelően egybefüggő körvonalat eredményez.

thresholdType-ként ’ THRESH_BINARY_INV’ módszert választottam, ami a küszöb alatti intenzitás értékekhez 255-öt, felette 0-át rendel. Ezáltal világosabb terület körül lévő sötétebb pixeleket emeli ki.



A C értékével lényegében megadhatjuk a pixelek intenzitása közötti elvárt különbséget. Jelen feladatnál ez a legfontosabb paraméter, hiszen alapfeltevés, hogy a pöttyök intenzitása nagyban eltér a kockáétól. Ezért a küszöb értéket C = 20 értékkel tolom el.

A fenti paraméterekkel a háttér teljesen fekete lesz és a pöttyök körvonala jól kivehető.

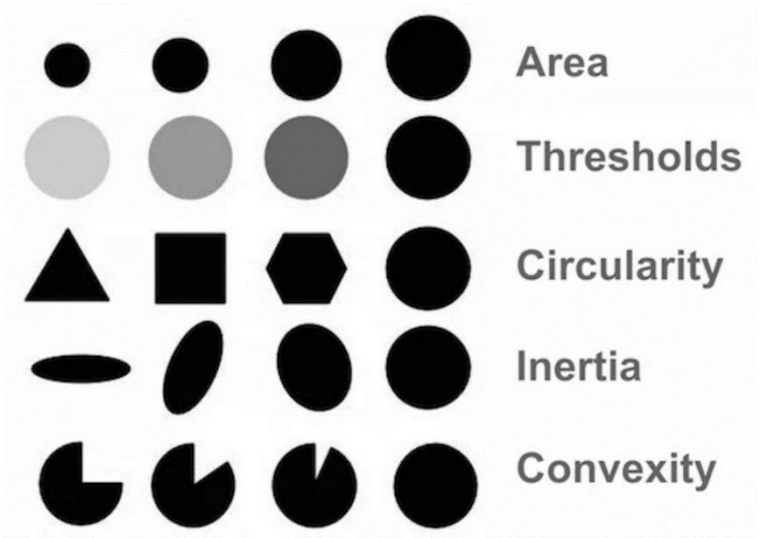
A módszer elég egyszerű, azonban a clusterzésnél gondot jelenthet, hogy a fehér pöttyök külső kerületét emeljük ki, míg fekete pöttyöknél a belsőt, ezáltal a körök átmérője függ annak eredeti színétől.

A tesztek során ez nem okozott problémát.

2.2. SimpleBlobDetector

OpenCV SimpleBlobDetector algortimusa a képen egybefüggő, hasonló intenzitású területeket keres. Mivel előző lépéssel a pöttyök körvonalát megjelölük, így ezzel a függvénnyel a körvonalon belüli fekete részt könnyen megkereshetjük.

A SimpleBlobDetector paraméterei között megkötéseket adhatunk a ’blob’ méretére, színére, és formájára:



Jelen esetben a méret csak a zajok további szűrése miatt érdekes.

Thresholds paraméter szintén kevésbé lényeges.

A fontos megkötések a formára vonatkoznak ugyanis a pöttyök erősen kör alakúak így ezeknek a paramétereknek (Circularity, inertia, convexity) 1 közeli értéket adtam.

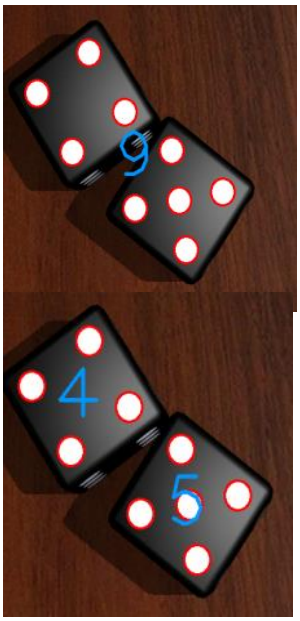
2.3. DBscan

DBscan koordináta, vagy távolságmátrix alapján keres egymáshoz közel álló pontokat, és ezeket csoportosítja egy-egy clusterbe. Jelen feladatban DBscan ideális, hiszen ennek a függvénynek nem kell előre megadni az elvárt clusterek számát. Mivel a képeken a kockák mennyisége változik, ezért ez az érték előre nehezen meghatározható.

*sklearn.cluster.DBSCAN(eps=0.5, *, min_samples=5, metric='euclidean', metric_params=None, algorithm='auto', leaf_size=30, p=None, n_jobs=None)*

Fontos paraméterei az ’eps’, ami az egy clusterbe sorolandó pontok közötti maximális távolságot adja meg, illetve a min_samples, ami pedig a clusterek minimális elemszámát rögzíti.

Utóbbit minden esetben 1 nek vettem, így a képen lévő összes felismert pont clusterbe lesz sorolva.



A kockák azonosításánál kulcsfontosságú az eps értéke. Ezt a felismert Blob-ok közül a legnagyobb átmérőjéhez kapcsoltam, ezáltal a nagyítás okozta problémák teljesen kiköszübölhetőek.

A 2-es oldalon a pöttyök távolsága elég nagy, így előfordulhat, hogy két egymáshoz közel lévő kocka egy clusterbe kerül helyes eps megválasztás esetén is.

Ennek kiküszöbölésére ’lépcsős’ clusterezést dolgoztam ki, amely fokozatosan növeli a pontok között megengedett távolságot. Ezzel a módszerrrel előbb elkülöníthetőek a 6-os, majd 3-as és 5-ös, 4-es és végül a 2-es 1-es kockák (pontok távolsága alapján).

Ez jelentősen javítja az algoritmus pontosságát.

3. Részletes leírás

A következő fejezetekben részletesen bemutatom a program egyes részeit, forráskód feltüntetésével.

3.1. Inicializálás

Indításkor két paraméterre van szükségünk a felhasználótól a változók előkészítéséhez. Az egyik természetesen a feldolgozandó képek elérési útja.

Ezt rögtön induláskor bekérjük, és a ’path_in’ változóban tároljuk. Ebben a mappában lévő .png kiterjesztésű képek elérési útját pedig a filelist array-be mentjük. A képek feldolgozása során ezen a listán iterálunk végig.

Az eredmények minden esetben ugyanezen mappában a Result.csv fájlba íródnak. Az eredményeket 'writer' objektumon keresztül írjuk az eredményfájlba.

file main.py
function Init(chk)

```
6   def Init():
7       print('Specify folder:')
8
9       global path_in
10      path_in = input()
11      if path_in == '':
12          path_in = 'C:/Users/acer/Pictures/Pictures/'
13
14      global path_out
15      path_out =  path_in + "Result.csv"
16
17      global filelist
18      try:
19          filelist = [f for f in os.listdir(path_in) if f.endswith(".png")]
20      except:
21          exit()
22
23      global pic_count
24      pic_count = len(filelist)
25      print("Count_of_images: " + str(pic_count))
26      if pic_count == 0:
27          exit()
28
29      global writer
30      writer = csv.writer(open(path_out,'w',newline = ''))
31      print('Writing: ' + path_out)
32      print()
    ...
```

A második paraméter egy boolean, ami azt tárolja, hogy a felhasználó szeretné-e az algoritmus eredményét megjeleníteni és ellenőrizni futás közben.

A választól a program több funkciója is függ, ugyanis ha nem, akkor nincs szükség az eredmény megjelítésére és annak ellenőrzésére sem (statisztikák sem készülnek). Ilyenkor elég csak a pöttyök felismerését és a clusterezést elvégezni, majd az eredményeket csv fájlba írni.

Ha a felhasználó szeretné látni az eredményt akkor elő kell készíteni a fenti funkciókhoz szükséges változókat is.

3.1.1. Eredményfájl előkészítése

Manuális validálás esetén az eredményfájl egyel több oszlopot kell, hogy tartalmazzon. Így a headert ennek megfelelően módosítani kell.

A válasz rögzítése és a statisztikák készítése érdekében 'chk' tól függően létrehozok két dictinary-t is, amik futás közben segítik a felhasználó válaszának értelmezését.

file main.py
function Init()

```
...
    print('Validate results? y/n')
    global chk
    chk = chr(msvcrt.getch()[0]) == 'y'
    print(chk)

    header = ['Picture', 'Number_of_dice', 'Results']

    if chk:
        header.insert(1,'Response')

        global response_dic
        response_dic = {
        'y':"correct",
        'd':"dot",
        'c':"clustering",
        'b':"both"}

        global stats_dic
        stats_dic = {
        "correct":0,
        "dot":0,
        "clustering":0,
        "both":0,
        "unknown":0}

        print('Press y/d/c/b to validate result.')
```

Az inicializálást követően kiirom a talált képek darabszámát, és az eredményfájlba írom ennek, valamint a header változónak az értékét.

3.2. Fő ciklus

A képek listájának előkészítése után. a fő ciklusban az elérési utak listáján iterálok, és a image.py Process_img függvényének segítségével feldolgozzuk a talált képeket. Az eredények értelmezéséért a PrepResults függvény felel.

chk-tól függően a ciklusban vizuálisan, és a konzolon is megjelenítjük az eredményt az Overlay és a CheckResult függvényekkel, és bekérjük az esetlegesen ejtett hiba minőségét. (clustering/dot error)

Az eredményeket writeren keresztül a csv fájl új sorába írjuk. A ciklus végén, ha készült statisztika, azt is az eredmény fájlba írjuk.

file main.py
function

```
...
for f in filelist:
    print('Processing: ' + f)

    dots, dice = image.Process_img(path_in+f)
    res = PrepResults(path_in+f, dice)

    if chk:
        image.Overlay(path_in+f, dice, dots)
        res = CheckResult(res)
        image.Close()

    writer.writerow(res)

if chk:
    for key in stats_dic:
        print(key + ': ' + str(stats_dic[key]))
        writer.writerow([key + ':,' + str(stats_dic[key])])
```

3.2.1. *Futás közbeni ellenőrzés*

A CheckResult függvény karakteresen megjeleníti a felismert kockák darabszámát, és az azokon lévő pöttyöket, majd bekéri a felhasználótól az eredmény helyességét. A válasz az eredmény sorban (res), és a stat_dic-ben tárolódik.

Correct?, y = correct, d = dot error, c = clustering error, b = both

file main.py
function CheckResult(res)

```
def CheckResult(res):
    print(res)
    print('Correct?, y = correct, d = dot error, c = clustering error, b = both')

    resp = response_dic.get(chr(msvcrt.getch()[0]),"unknown")
    print(resp)
    print()
    stats_dic[resp] += 1/pic_count
    res.insert(1,resp)

    return res
```

3.3. Képek feldolgozása

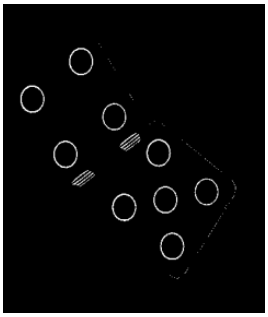
A ciklus során a képeket sorra, egyesével dolgozzuk fel. első lépésként az elérési út segítségével megnyitjuk őket, majd ’előkezelést’ követően a pöttyök felismerése, végül azok csoportosítása következik.

file image.py
function Process_img(pic)

```
def Process_img(pic):
    img = OpenImage(pic)
    BW_img = PreProcess(img)
    dots = GetDots(BW_img)
    dice = GetDice(dots)

    return dots, dice
```

3.3.1. *Kép előkészítés*



Az előkészítés során előbb szürkeárnyalatos képpé, végül adaptív gauss küszöböléssel fekete-fehér képpé alakítjuk a beolvasott képeket.

Mivel a kockákon a pöttyök jól elkülönülnek a kocka színétől, ezzel a módszerrel a pöttyök éle egyértelműen kiemelhető.

Az előkészített képen előbb a pöttyöket keressük meg, majd a pöttyöket csoportosítva a kockákat azonosítjuk.

file image.py
function OpenImage(path)

```
28  def OpenImage(path):
29      img = cv2.imread(path,1)
30
31      if img is None:
32          print("Error: Cannot read image: ", path)
33          return -1
34
35      return img
```

file image.py
function PreProcess(img)

```
37  def PreProcess(img):
38      Gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
39      BW_img = cv2.adaptiveThreshold(Gray_img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,\
40              cv2.THRESH_BINARY_INV,9,20)
41
42      if BW_img is None:
43          print("Error: Cannot process image")
44          return -1
45
46      return BW_img
```

3.4. Pöttyök felismerése

A pontokat OpenCV SimpleBlobDetectorral keresem. A függvény paraméterei között a pötty formájára, korábban említett megkötéseket tehetünk. Ezáltal ideális a kockákon lévő többnyire kör alakú pöttyök felismerésére.

Az eredményeket látszik, hogy helyes paraméterezéssel a pöttyök 100% pontossággal kerültek felismerésre minden képen.

A pöttyöket a 'dots' array-ben tárolom, amelynek egy eleme a pöttyök közepének koordinátáit, a pötty méretét, és még egyéb (jelenleg lényegtelen) tulajdonságait tartalmazza.

file image.py
function

```
...
#detector params----------------------------------
params = cv2.SimpleBlobDetector_Params()

params.minThreshold = 0
params.maxThreshold = 255

params.filterByArea = True
params.minArea = 10

params.filterByCircularity = True
params.minCircularity = 0.6

params.filterByConvexity = True
params.minConvexity = 0.8

params.filterByInertia = True
params.minInertiaRatio = 0.8

detector = cv2.SimpleBlobDetector_create(params)
```

file image.py
function GetDots(BW_img)

```
def GetDots(BW_img):
    dots = detector.detect(BW_img)
    return dots
```

3.5. Kockák csoportosítása

A pöttyök csoportosítását DBscan módszerrel végeztem, ugyanis ennek a clusterezésnek nem kell megadni a clusterek számát előre. Mivel a kockák száma a képen ismeretlen, ezért a clusterek számát sem tudhatjuk előre.

DBscan a pontok távolsága alapján csoportosítja azokat. A paraméterek között eps értéke az egy csoportba tartozó pontok közötti távolságot határozza meg.

Mivel a képek eltérő nagyításúak, ez a távolsák a pöttyök méretétől függ.

Eps-t tehát dinamikusan, a képen felismert pöttyök közül a legnagyobb méretének függvényében határozom meg. Tapasztalat szerint a 6-os oldalankét szomszédos pötty távolsága a pötty 1.4-szerese. Ezt az értéket a 'dist' változóban tárolom.

A csoportosításhoz ezután csak a pöttyök helyzete szükséges. Ezt a dots_next array tartalmazza. A dots_6 és dots_rest array-ek a lépcsős clusterezéshez szükségesek.

A clusterezés eredménye, vagyis a kockák a dice array-be íródnak. Ez az array kockánként annak középpontját, és a rajta lévő pöttyök számát tartalmazza.

file image.py
function GetDice(dots)

```
84  def GetDice(dots):
85      dots_next = []
86      S = []
87
88      for d in dots:
89          if d != None:
90              dots_next.append(d.pt)
91              S.append(d.size)
92
93      dist = max(S)*SizeFactor
94      dots_next = np.asarray(dots_next)
95
96      dots_6 =[]
97      dots_rest = []
98      dice = []
    ...
```

3.5.1. *Egyszerű csoportosítás*

Az egyszerű csoportosításnál a pöttyök közötti legnagyobb távolság alapján csoportosítok, azaz a 2 es oldalon lévő két átellenes pötty közötti távolság az eps paraméter értéke.

A DBscan a cluster sorszámát jellemző label értéket rendel a pontokhoz. Ez alapján a ciklusban az egyes clusterekbe tartozó pöttyök a 'die' változóba menthetőek.

Ez a módszer, az egy kockán lévő pontokat biztosan egy csoportba sorolja, azonban egymáshoz közel lévő kockák esetén két kockát is egybe vehet.

file image.py
function Simple_cluster(dots, dice, Factor)

```
52  def Simple_cluster(dots, dice, Factor):
53      if len(dots) > 0:
54
55          clustering = cluster.DBSCAN(eps=Factor, min_samples=1).fit(dots)
56
57          num_dice = max(clustering.labels_) + 1
58          for i in range(num_dice):
59              die = dots[clustering.labels_ == i]
60              centroid = np.mean(die, axis=0)
61              dice.append([len(die), *centroid])
62      return dice
```



Ez elég sok clusterezési hibához vezethet. Ennek elkerülésére a lépcsős clusterezési módszert használtam, ami fokozatosan növeli eps értékét. Ezt mutatom be a következő fejezetben.

3.5.2. *Lépcsős csoportosítás*

A lépcsős módszer szintén DBscan függvényen alapul. Az egyetlen változás, hogy itt összesen 5, kétféle (separeting és simple) clusterezési ciklus fut le.

A szeparáló clusterezés annyiban más, hogy kritériumként megadható, hogy pontosen hány pötty legyen egy clusterben. Ezáltal felismerhető például a csak 3-as oldal és elkülöníthető a pöttyökből. A dots_rest a maradék pontokat tartalmazza, amin aztán újabb clusterezés végezhető.

file	image.py
function	Separating_cluster(dots, dice, Factor, criteria)

```
64  def Separating_cluster(dots, dice, Factor, criteria):
65      dots_rest = []
66      if len(dots) > 0:
67
68          clustering = cluster.DBSCAN(eps=Factor, min_samples=1).fit(dots)
69
70          num_dice = max(clustering.labels_) + 1
71          for i in range(num_dice):
72              die = dots[clustering.labels_ == i]
73
74              if len(die) in criteria:
75                  centroid = np.mean(die, axis=0)
76                  dice.append([len(die), *centroid])
77              else:
78                  for d in die:
79                      dots_rest.append(d)
80
81          dots_rest = np.asarray(dots_rest)
82      return dice, dots_rest
```

A 6 os oldal elkülönítése két lépést igényel, mégis célszerű ezzel az oldallal kezdeni, ugyanis az egy ’oldalfélen’ lévő pöttyök közötti távolság itt a legkisebb. Ezt kihasználva a 6-os oldal biztosan elkülöníthető az összes többitől.

Korábban említettem, hogy a legközelebbi pontok távolsága tapasztalat alapján a pöttyök méretének 1.4 szerese, így első lépésben ezzel az értékkel végzünk egyszerá clusterezést, a 6-os oldalokon lévő 3-3 pöttyök csoportosítása érdekében. Ezek a pöttyök a többinél közelebb állnak egymáshoz, így a maradék pötty (ami nem 6-os oldalon van) a dots_rest ben marad.

A 6-os oldal ekkor azonban 2 clusterből áll még, így ezeken a pöttyökön egy újabb simple clusterezést végzek az összevonáshoz.

Ezzel a két clusterezéssel tehát kizárólag a 6-os oldalú kockákat szűrtem ki a többi közül.

file image.py
function GetDice(dots)

```
...
#6-------------------------------------------------------------------------
    if len(dots_next) > 0:
        clustering = cluster.DBSCAN(eps=dist, min_samples=1).fit(dots_next)

        num_dice = max(clustering.labels_) + 1
        for i in range(num_dice):
            die = dots_next[clustering.labels_ == i]

            if len(die) == 3:
                for d in die:
                    dots_6.append(d)
            else:
                for d in die:
                    dots_rest.append(d)

        dots_6 = np.asarray(dots_6)
        dots_next = np.asarray(dots_rest)

        dice = Simple_cluster(dots_6, dice, dist*2)
...
```

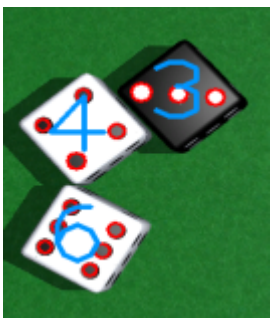
Ezután a fenti Separating_cluster függvénnyel elkülönítem a 3-as 5-ös kockákat (második legkisebb távolság a pöttyök között, madj a 4-es kockákat.

A kettesek csoportosítása a maradék pontokból az előző fejezetben leírt Simple_cluster -rel történik.

Ezzel a clusterezési hibák nagyobbik része kiszűrhető, de közel lévő 4 es vagy 2 es kockákat ez a módszer sem tud elkülöníteni

file image.py
function GetDice(dots)

```
...
#35-------------------------------------------------------------------------
    dice, dots_next =  Separating_cluster(dots_next, dice, dist*1.4, [3,5])
#4--------------------------------------------------------------------------
    dice, dots_next =  Separating_cluster(dots_next, dice, dist*1.9, [4])
#rest-----------------------------------------------------------------------
    dice = Simple_cluster(dots_next, dice, dist*2.7)
        dice = Simple_cluster(dots_6, dice, dist*2)
    return dice
```



.

3.6. Eredmény grafikus megjelenítése

A grafikus megjelenítés a manuális ellenőrzés miatt lényeges. A pöttyöket piros körvanallal jelölöm, a kockák középpontjára, pedig az oldalhoz rendelt pöttyök számát jelenítem meg.

A fenti képeken ez látható is. A képet kicsinyített formában jelenítem meg a képernyőn. A felhasználó válasza után pedig bezárom.

file image.py
function Overlay(pic,dice,dots)

```
136 def Overlay(pic,dice,dots):
137     img = OpenImage(pic)
138 
139     for d in dots:
140         pos = d.pt
141         r = d.size / 2
142 
143         cv2.circle(img,                         #image
144                    (int(pos[0]),int(pos[1])), #position
145                     int(r),                    #radius
146                    (0, 0, 255),                #BGR color
147                     2)                         #thickness
148 
149     for d in dice:
150         textsize = cv2.getTextSize(str(d[0]), cv2.FONT_HERSHEY_PLAIN, 6, 4)[0]
151         cv2.putText(img, str(d[0]),
152                     (int(d[1] - textsize[0] / 2),int(d[2] + textsize[1] / 2)),
153                     cv2.FONT_HERSHEY_PLAIN, 6, (255, 150, 0), 4)
154 
155     cv2.imshow("image",cv2.resize(img,[400,700],interpolation = cv2.INTER_AREA))
156     cv2.waitKey(100)
```

3.6.1. Eredmény fájlba írása

Az eredményeket egy Results.csv fájla írom az alábbi formátumban:

Az első sor a képek számát tartalmazza, ezt követi egy header, majd az egyes képek eredményei. Pl.:

Count_of_images:;109
Picture;Response;Number_of_dice;Results
C://Users/acer/Pictures/Pictures/Screenshot_20221113-162407.png;correct;4D;3;5;4;1

A második oszlop 'Response' a chk (manuális ellenőrzés) től függően kerül beillesztésre.

Az utolsó sorok pedig a statisztikákat tartalmazzák (ha készült) pl.:

correct:;0,76146789
dot:;0
clustering:;0,23853211
both:;0
unknown:;0

’;’ mentén szeparálva az oszlopokat az eredmények további feldolgozása lehetséges.

4. Tesztelés

A tesztet két módszerrel is elvégeztem, az egyik a pontok egyszerű clustereléssel való csoportosítását végzi. Ennek eredményei a Result_simple fájlban látható.

Később az algoritmust "lépcsős" clustereléssel egészítettem ki, Ennek eredménye a Result_Complex fájl mutatja be.

A feltöltött forráskód csak a "lépcsős" módszert tartalmazza de igen egyszerűen átalakítható az egyszerű működésre.

A SimpleBlolbDetection és a Dbscan használatával a dobókockák elég jó pontossággal ismerhetőek fel. A pöttyök, a fenti módon paraméterezett Gauss küszöbölés, és SimpleBlolbDetector használatával 100%-os pontossággal meghatározhatóak.

Az egyszerű DBscan módszer a clusterezést a 109 képen 76% pontossággal végezte el. Ezzel szemben a lépcsős clusterezés ugyanezen a tesztmappán 95% pontosságot ért el.

Lásd Result_Simple és Result_Complex fájlokban.

Number of pictures: 109			
Simple		Complex	
correct:	76%	correct:	95%
dot:	0	dot:	0
clustering:	24%	clustering:	5%
both:	0	both:	0
unknown:	0	unknown:	0

5. Fejlesztési lehetőségek

Jelenleg a pöttyök felismerésénél egy-egy pont mérete függ annak színétől. Ezt az adaptív küszöbölés eltérő paraméterezésével, vagy éldetektálási módszerekkel elkerülhetjük, azonban a tesztmappán ez a hiba nem volt kimutatható.

A pöttyök méretének pontosabb meghatározásával a clusterezés is javítható lehet. Itt használható lehet a távolságmátrix alapján működő algoritmus is, ahol a legkisebb távolságot a távolságmátrix minimumával közelítjük. Ekkor ügyelni kell arra, hogy a képen nem biztos, hogy van 6-os oldal.

A kockák azonosítása lehetséges akár éldetektálással majd téglalapok keresésével is.

6. Felhasználói leírás

A program indulásakor a konzolon meg kell adni a vizsgálni kívánt képek elérési útját. Például:

```
Specify folder:
C:/Users/acer/Pictures/Pictures/
```

A program az ebben a mappában lévő .png kiterjeszétű fájlokon fog iterálni. Ezeken a képeken keresi a dobókockákon lévő pöttyöket és magukat a kockákat.

Ezt követően információt kapunk a talált képek darabszámáról, és az eredményfájl elérési útjáról:

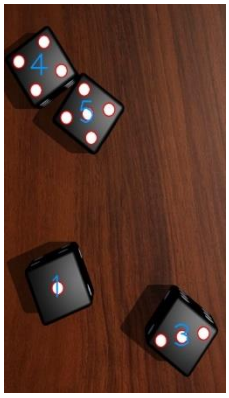
```
Specify folder:
C:/Users/acer/Pictures/Pictures/
Count_of_images: 109
Writing: C:/Users/acer/Pictures/Pictures/Result.csv
```

A következő kérdés a manuális ellenőrzésre vonatkozik:

```
Validate results? y/n
False
```

```
Validate results? y/n
True
```

y megnyomásával futás közben ellenőrzést végezhetünk. Ekkor a program grafikusan és karakteresen is megjeleníti az eredményeket, és lehetőségünk van értékelni azokat az 'y' 'c' 'd' 'b' gombokkal. Például:



```
Press y/d/c/b to validate result.
Processing: Screenshot_20221113-162407.png
['C:/Users/acer/Pictures/Pictures/Screenshot_20221113-162407.png', '4D', 3, 5, 4, 1]
Correct?, y = correct, d = dot error, c = clustering error, b = both
correct
```

Futás közbeni ellenőrzésnél a program rögzíti a válaszainkat és statisztikát is készít a hibákról.

Ha futás közben nem akarjuk ellenőrizni az eredményt, akkor annak pontossága csak utólag manuálisan értékelhető ki.

```
Validate results? y/n
False

Processing: Screenshot_20221113-162407.png
Processing: Screenshot_20221113-162417.png
Processing: Screenshot_20221113-162437.png
Processing: Screenshot_20221113-162445.png
Processing: Screenshot_20221113-162453.png
Processing: Screenshot_20221113-162538.png
```

Az eredményeket soronként Result.csv tartalmazza az alábbi formában:

Count_of_images:;109
Picture;Response;Number_of_dice;Results
C://Users/acer/Pictures/Pictures/Screenshot_20221113-162407.png;correct;4D;3;5;4;1

A statisztikák az eredményfájl végén találhatóak, formátumuk (ha készült) például:

correct:;0,76146789
dot:;0
clustering:;0,23853211
both:;0
unknown:;0

';' mentén szeparálva az oszlopokat az eredmények további feldolgozása lehetséges.

7. Források

1. https://gist.github.com/qgolsteyn/261289d999a8d6288ce8c0b8472e5354
2. https://www.mathworks.com/matlabcentral/answers/248036-how-to-identify-black-dots-on-dice-via-image-processing-matlab
3. https://www.davidepesce.com/2019/09/06/dice-reader-part-1/
4. https://golsteyn.com/writing/dice
5. https://github.com/sujanay/Dice-Dot-Counter
6. https://docs.opencv.org/4.x/d7/d1b/group__imgproc__misc.html#ga72b913f352e4a1b1b397736707afcde3
7. https://docs.opencv.org/4.x/d7/d1b/group__imgproc__misc.html#gaa42a3e6ef26247da787bf34030ed772c
8. https://pyimagesearch.com/2021/05/12/adaptive-thresholding-with-opencv-cv2-adaptivethreshold/
9. https://learnopencv.com/blob-detection-using-opencv-python-c/
10. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html

8. Forráskód

file main.py

```
1   import os
2   import msvcrt
3   import csv
4   import image
5
6   def Init():
7       print('Specify folder:')
8
9       global path_in
10      path_in = input()
11      if path_in == '':
12          path_in = 'C:/Users/acer/Pictures/Pictures/'
13
14      global path_out
15      path_out =  path_in + "Result.csv"
16
17      global filelist
18      try:
19          filelist = [f for f in os.listdir(path_in) if f.endswith(".png")]
20      except:
21          exit()
22
23      global pic_count
24      pic_count = len(filelist)
25      print("Count_of_images: " + str(pic_count))
26      if pic_count == 0:
27          exit()
28
29      global writer
30      writer = csv.writer(open(path_out,'w',newline = ''))
31      print('Writing: ' + path_out)
32      print()
33
34      print('Validate results? y/n')
35      global chk
36      chk = chr(msvcrt.getch()[0]) == 'y'
37      print(chk)
38
39      header = ['Picture', 'Number_of_dice', 'Results']
40
41      if chk:
42          header.insert(1,'Response')
43
44          global response_dic
45          response_dic = {
46          'y':"correct",
47          'd':"dot",
48          'c':"clustering",
49          'b':"both"}
50
51          global stats_dic
52          stats_dic = {
53          "correct":0,
54          "dot":0,
55          "clustering":0,
56          "both":0,
57          "unknown":0}
58
59          print('Press y/d/c/b to validate result.')
```

```
60
61      writer.writerow(["Count_of_images:," + str(pic_count)])
62      writer.writerow(header)
63      print()
64  #-------------------------------------------------------------
65
66  def PrepResults(pic, dice):
67      row=[d[0] for d in dice]
68      row.insert(0,str(len(dice)) + 'D')
69      row.insert(0,pic)
70      return row
71
72  def CheckResult(res):
73      print(res)
74      print('Correct?, y = correct, d = dot error, c = clustering error, b = both')
75
76      resp = response_dic.get(chr(msvcrt.getch()[0]),"unknown")
77      print(resp)
78      print()
79      stats_dic[resp] += 1/pic_count
80      res.insert(1,resp)
81
82      return res
83
84  #-------------------------------------------------------------
85
86  Init()
87
88  for f in filelist:
89      print('Processing: ' + f)
90
91      dots, dice = image.Process_img(path_in+f)
92      res = PrepResults(path_in+f, dice)
93
94      if chk:
95          image.Overlay(path_in+f, dice, dots)
96          res = CheckResult(res)
97          image.Close()
98
99      writer.writerow(res)
100
101 if chk:
102     for key in stats_dic:
103         print(key + ': ' + str(stats_dic[key]))
104         writer.writerow([key + ':,' + str(stats_dic[key])])
```

file image.py

```
1   import cv2
2   import numpy as np
3   from sklearn import cluster
4
5   #detector params----------------------------------
6   params = cv2.SimpleBlobDetector_Params()
7
8   params.minThreshold = 0
9   params.maxThreshold = 255
10
11  params.filterByArea = True
12  params.minArea = 10
13
14  params.filterByCircularity = True
```

```
params.minCircularity = 0.6

params.filterByConvexity = True
params.minConvexity = 0.8

params.filterByInertia = True
params.minInertiaRatio = 0.8

detector = cv2.SimpleBlobDetector_create(params)

SizeFactor = 1.4

#-----------------------------------------------------
def OpenImage(path):
    img = cv2.imread(path,1)

    if img is None:
        print("Error: Cannot read image: ", path)
        return -1

    return img

def PreProcess(img):
    Gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    BW_img = cv2.adaptiveThreshold(Gray_img,255,cv2.ADAPTIVE_THRESH_GAUSSIAN_C,\
            cv2.THRESH_BINARY_INV,9,20)

    if BW_img is None:
        print("Error: Cannot process image")
        return -1

    return BW_img

def GetDots(BW_img):
    dots = detector.detect(BW_img)
    return dots

def Simple_cluster(dots, dice, Factor):
    if len(dots) > 0:

        clustering = cluster.DBSCAN(eps=Factor, min_samples=1).fit(dots)

        num_dice = max(clustering.labels_) + 1
        for i in range(num_dice):
            die = dots[clustering.labels_ == i]
            centroid = np.mean(die, axis=0)
            dice.append([len(die), *centroid])
    return dice

def Separating_cluster(dots, dice, Factor, criteria):
    dots_rest = []
    if len(dots) > 0:

        clustering = cluster.DBSCAN(eps=Factor, min_samples=1).fit(dots)

        num_dice = max(clustering.labels_) + 1
        for i in range(num_dice):
            die = dots[clustering.labels_ == i]

            if len(die) in criteria:
                centroid = np.mean(die, axis=0)
                dice.append([len(die), *centroid])
            else:
```

```
                for d in die:
                    dots_rest.append(d)

        dots_rest = np.asarray(dots_rest)
    return dice, dots_rest

def GetDice(dots):
    dots_next = []
    S = []

    for d in dots:
        if d != None:
            dots_next.append(d.pt)
            S.append(d.size)

    dist = max(S)*SizeFactor
    dots_next = np.asarray(dots_next)

    dots_6 =[]
    dots_rest = []
    dice = []

#6-------------------------------------------------------------------------
    if len(dots_next) > 0:
        clustering = cluster.DBSCAN(eps=dist, min_samples=1).fit(dots_next)

        num_dice = max(clustering.labels_) + 1
        for i in range(num_dice):
            die = dots_next[clustering.labels_ == i]

            if len(die) == 3:
                for d in die:
                    dots_6.append(d)
            else:
                for d in die:
                    dots_rest.append(d)

        dots_6 = np.asarray(dots_6)
        dots_next = np.asarray(dots_rest)

        dice = Simple_cluster(dots_6, dice, dist*2)

#35------------------------------------------------------------------------
    dice, dots_next =  Separating_cluster(dots_next, dice, dist*1.4, [3,5])
#4-------------------------------------------------------------------------
    dice, dots_next =  Separating_cluster(dots_next, dice, dist*1.9, [4])
#rest----------------------------------------------------------------------
    dice = Simple_cluster(dots_next, dice, dist*2.7)
    return dice

def Process_img(pic):
    img = OpenImage(pic)
    BW_img = PreProcess(img)
    dots = GetDots(BW_img)
    dice = GetDice(dots)

    return dots, dice

def Overlay(pic,dice,dots):
    img = OpenImage(pic)

    for d in dots:
        pos = d.pt
```

```
140         r = d.size / 2
141 
142         cv2.circle(img,                         #image
143                    (int(pos[0]),int(pos[1])), #posistion
144                     int(r),                     #radius
145                    (0, 0, 255),                #BGR color
146                     2)                         #thickness
147 
148     for d in dice:
149         textsize = cv2.getTextSize(str(d[0]), cv2.FONT_HERSHEY_PLAIN, 6, 4)[0]
150         cv2.putText(img, str(d[0]),
151                     (int(d[1] - textsize[0] / 2),int(d[2] + textsize[1] / 2)),
152                     cv2.FONT_HERSHEY_PLAIN, 6, (255, 150, 0), 4)
153 
154     cv2.imshow("image",cv2.resize(img,[400,700],interpolation = cv2.INTER_AREA))
155     cv2.waitKey(100)
156 
157 def Close():
158     cv2.destroyAllWindows()
159 
```