MED3: King dominos mini-projekt

1 Introduktion

Det her projekt omkring billedbehandling. Formålet med projektet har været at udforske forskellige muligheder for vha. billedbehandling at lavet et system der er i stand til at tælle point op for en spilleplade i brætspillet King Dominos.

1.1 En gameplan

Figur 1: Spilleplade 7

Da pointene på hver plade er udregnet ved at tage mængden af sammenhængene, ligedan miljøområder, ganget med antallet af kroner inden for samme, sammenhængende miljøområde, valgte vi at gennemgå pladerne for at se hvilke udfordringer dette måtte indebære. Der findes et mønster i, at alle de felter med kroner på, inden for hvert miljø, har sit eget særpræg i form af forskellige bygninger. Som ses på figur 1 af spilleplade 7, burde der findes en krone på pladen, på den markerede brik én plads nord for centrum. Da denne ikke er synlig grundet at den placerede bygning blokerer for udsynet til denne, valgte vi at udelukke en metode, hvor vi ville bruge kronerne til at identificere hvor meget et felt er værd.

I stedet har vi testet et væld af forskellige fremgangsmåder for at se, hvad der fungerede bedst til at kende forskel på hvert enkelt felt af hver af de forskellige typer. Der findes i alt 16 forskellige felter som vi har navngivet som vist i følgende tabel, tabel 1:

mine_0	grass_0	waste_0	desert_0	forest_0	water_0
mine_1	grass_1	waste_1	desert_1	forest_1	water_1
mine_2	grass_2	waste_2			
mine_3					

Tabel 1: Brik typer

Navnet forrest i hver kolonne repræsenterer typen af miljø, og tallet der følger repræsenterer hvor mange kroner, der er på det givne felt. Forskellige miljøtyper kan have op til et forskelligt antal kroner. For hvert felt samlede vi 10 forskellige udsnit af billeder tilsvarende felter som vi gemte i en mappe opkaldt efter hver type. Dette agerede som vores test-materiale for vores program, hvorfra tanken var at programmet skulle sammenligne features beregnet på hvert test-tile med de tiles der senere hen ville blive fundet fra de spilleplader som programmet skulle udføre tests på. Eksempler på sådanne udsnit ses her:

Figur 2: Eksempler på udsnit fra forskellige miljø/krone-kombinationer

2. Implementering

2.1 Opdeling af billedet i slices

For at programmet skulle kunne sammenligne felterne i den givne spilleplade med vores testdata, opdelte vi først hvert billede i 25 slices, således at hver spillebrik, blev en slice for sig for at kunne klassificere hver brik for sig. Dette gjorde vi med følgende metode som vi har kaldt `sliceROI`:

```
def sliceROI(roi):
    output = []

    for y in range(0, roi.shape[0], int(roi.shape[0] / 5)):
        y_line = []
        for x in range(0, roi.shape[1], int(roi.shape[1] / 5)):
            slice = roi[y: y + int(roi.shape[0] / 5), x:x + int(roi.shape[1] / 5)].copy()
            y_line.append(slice)
        output.append(y_line)

    return output
```

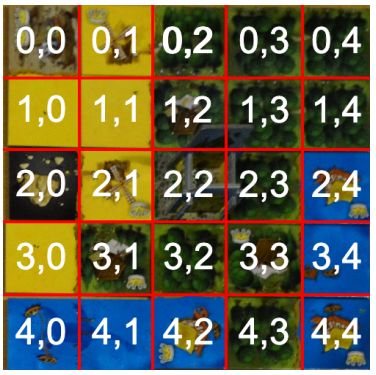
Code-listing 1: metode, sliceROI

Tanken med navnet var at vi senere ville kalde selve spillepladen for vores ROI, når den blev fundet blandt billederne der indeholder tegneserier.

Metoden opretter et array kaldet `output`. Derefter loopes der meget simpelt hen over y-aksen på den givne spilleplade, med en step-size svarende til y-aksens længde delt med fem. Da hvert spilleplade-billede har dimensionerne 500 x 500 pixels, svarer det til at metoden looper fra 0 til 500 med en step size på 100, så den looper frem gange. For hvert skridt i metoden oprettes et array beregnet til at indeholde hver slice for den tilsvarende

y-koordinat. Derefter loopes der på samme måde hen over x-aksen fra 0 til 500 med en step size på 100, og for hvert skridt laves et nyt slice svarende til den y-værdierne mellem nuværende y-koordinat til samme værdi plus 100, og på samme måde for x-aksen. På den måde får vi en slice for hver brik, som illustreret i figur 2.a. På figur 2.b ses også de resulterende koordinater som hvert slice tilgås på, via arrayet `output` med indicierne `[y][x]`. Fx vil et kald til `output[2][4]`, efter loopet har kørt, returnere det slice, der har det tilsvarende koordinat på figur 2.b.

Figur 2.a: Spilleplade 7, sliced



Figur 2.b: Spilleplade 7, sliced med tilsvarende koordinater

Ved at oprette et array med hver brik som en slice fra spillepladen har vi mulighed for at loope igennem dem alle, og derved lave udregninger på hver brik med det formål at sammenligne dem med vores test-data og derigennem forsøge at tildele hver brik en af de brik-typer som tidligere er blevet vist i tabel 1.

2.2 Opbevaring af test-data

De features, som vi til sidst endte med at vælge til at beskrive vores tiles bedst var at beregne gennemsnitsværdierne for hver farvekanal for hver tile. For at lyset ikke skulle påvirke farverne alt for meget, har vi taget kvadratroden af alle farverne for på den måde at gøre afstanden mellem lys og mørk mindre. Det vil altså sige at vi har en feature-vektor indeholdende tre værdier; `b_mean`, `g_mean` og `r_mean`. Disse beregnes i en metode vi har kaldt `calculateSliceColor_Mean`:

```
def calculateSliceColor_Mean(slice):
    b_mean = slice[:, :, 0].mean()
    g_mean = slice[:, :, 1].mean()
    r_mean = slice[:, :, 2].mean()

    return [b_mean, g_mean, r_mean]
```

Code-listing 2: calculateSliceColor_Mean

Da disse værdier, når beregnet på vores testdata altid vil være det samme valgte vi at gemme værdierne lokalt i filer, således at vi ikke behøvede at beregne dem hver gang. Disse værdier gemte vi i filer via en metode vi kaldte `saveTileVectorDictionary`. Denne

metode behøvede kun at køre en enkelt gang for hver gang vi ændrede den ønskede feature-vektor. Heri gemtes de 10 feature-vektorer for hver af de 10 test tiles, svarende til den tile-type de har. Så dictionariet indeholder 16 elementer, én for hver tile-type, hvor keyen i dicionariet er den tilsvarende tile-type og value er et array indeholdende de 10 feature-vektorer.
Efter at have gemt alle disse, kunne vi tilgå alle feature vektorerne i den oprettede via metoden `loadTileVectorDictionary`, der blot indlæser og returnerer dictionariet fra den gemte fil. Disse to metoder vises herunder i Code-listing 3. Heri er det array `tileTypes`, der loopes over blot et array af 16 string elementer, én for hver tile-type. I hver mappe er hver tile-slice navngivet med et tal mellem 0 og 9. Alle tile-slicene er også roteret 4 gange, da vi ville have lavet noget blob matching med midten, for at finde antallet af kroner på tile-slicen. Dette kunne vi desværre ikke få til at fungere, men derfor er der 4 mapper med 10 tiles i hver. Derfor kan man blot neste 1 for-loop, og loope over hver mappe, og herefter hvert billede, for at gemme alle features.

Den indlæste billedfil bruges i beregningen til at beregne en tilhørende feature-vektor med gennemsnitsværdier for hver farvekanal, som så gemmes i det array der tilhører tile-typen, hvorefter det array til sidst bliver gemt som valuen i dictionariet.

```python
def saveTileVectorDictionary():
    featureVectors = {}

    for tileType in tileTypes:
        tileArray = []
        for tile in range(10):
            img = cv.imread(f'King Domino dataset/Cropped_Tiles/{tileType}/{tile}.jpg')
            tileArray.append(np.array(calculateSliceColor_Mean(img)))
        featureVectors[tileType] = tileArray

    np.save(f'King Domino dataset/Cropped_Tiles/featureVectors', featureVectors)


def loadTileVectorDictionary():
    featureVectors = np.load(f'King Domino dataset/Cropped_Tiles/featureVectors.npy', allow_pickle=True)
    return featureVectors
```

Code-listing 3: metoderne til henholdsvis at gemme og indlæse vores feature-vektorer

2.3 Feature matching med test-data

Nu har vi vores test data. Det vi gerne vil nu er at sammenligne de samme værdier for hver slice med vores test data og se hvilken værdi den ligger tættest på via k-nearest neighbour. Det er så den vi går ud fra, der er den rigtige tile-type.

Måden vi tester det her på er at vi for hver tile kalder metoden `kNearestNeighbor`, der tager imod argumenterne `slice`, der er vores udsnit af brikken og `data`, der er det indlæste feature-vektor dictionary. Overordnet set går det ud på at kigger på afstandene mellem slice feature-vektoren og alle vektorer for vores test-data og så finder vi den nærmeste. Så vi bruger single-nearest neighbor, om man vil.

```python
def kNearestNeighbor(slice, data):
    sliceFeatureVector = calculateSliceColor_Mean(slice)

    distance, currentType = 20.0, 'None_0'

    for (tileType, tiles) in data.items():
        for tile in tiles:
            newDistance = calculateEuclidianDist(sliceFeatureVector, tile)
            if (newDistance < distance):
                distance = newDistance
                currentType = tileType

    return [currentType, distance]
```

Code-listing 4: metode til at beregne den mest nærliggende tile type.

I metoden, som også kan ses ovenover, har vi også to variable: `distance`, der er en `float` med en startværdi på `7`, og `currentType`, der er en `string` der har værdien `None_0`. Indmaden i metoden består i, at vi looper over dictionariet `data` og de arrays det indeholder. For hver feature-vektor fra vores data beregner vi afstanden (euclidian distance) mellem de to vektorer. Den afstand vi beregner gemmer vi i en variabel `newDistance`. Derefter sammenligner vi `newDistance` med `distance`. Hvis `newDistance` er mindre end `distance`, så ændres `distance` til den nye beregnede værdi fra `newDistance` og `currentType` ændres til en `string` svarende til den tile-type der hang sammen med den nye, kortere distance. Det gøres for alle tiles i vores data, så vi til sidst ender med den tile-type der har haft den korteste distance ud af de 640 tiles fra vores testdata. Den fundne type og afstanden returneres af metoden.

2.4 Opbevaring af tile-typer og distancer i arrays

Denne metode kaldes i en anden metode, `getAllSliceTypes`, der tager de returnerede værdier for både slice-typerne og distancerne for hver slice, og kommer dem i arrays med koordinater svarende til dem som hvert slice har i slice-arrayet. Det gøres ved blot at loope over først y og derunder x aksen af arrayet af slices, og på samme måde gemme typen og distancen i hvert deres array, som til sidst returneres

```python
def getAllSliceTypes(slices, data):
    sliceTypes = []
    distances = []

    for y, row in enumerate(slices):
        sliceTypeRow = []
        distanceRow = []

        for x, slice in enumerate(row):
            sliceType, distance = kNearestNeighbor(slice, data)
            sliceTypeRow.append(sliceType)
            distanceRow.append(distance)
        sliceTypes.append(sliceTypeRow)
        distances.append(distanceRow)

    return [sliceTypes, distances]
```

Code-listing 5: metode til at få slice-type arrays og distancer

Nu har vi altså bestemt hver slice type blandt de 16 typer fra tabel 1, og kan for syns skyld skrive det ovenpå det billede vi lavede slices ud af og så se, hvor godt programmet klarede det. Et eksempel på et resultatet kan ses herunder på spilleplade 8 på figur 3. I dette tilfælde, som det ses på figuren, har den alle brikker rigtige med undtagelse af (y,x) = (0,2), hvor den ikke mener der burde være en krone på feltet.

Figur 3: Spilleplade 8, resultat eksempel

2.5 Beregning af den endelige score

Nu har vi de formodede slice-typer for hver slices i billedet og tilhørende koordinater for hver slice. Vores måde at regne sammenhængende felter ud tager udgangspunkt i grassfire algoritmen. Algoritmen køres i dette tilfælde med en four-connectivity over hver tile-type i hele billedet for at finde sammenhængende blobs, i vores tilfælde, kaldet `islands`, hvor territorierne matcher. Ud fra de islands har vi også muligheden for at tælle antal kroner ved at kigge på tallet til sidst i tile-typen. Tallet samles i et array, `crownArray`, ved brug af metoden `split`, hvor den bedes dele hver tile-type `string` i hver `island` på hver side af

"_". For at beregne den endelige score finder vi først summen af `crownArray`, hvilket giver os antallet af kroner i den givne `island`. Det tal ganger vi så med `len(islands)`, antallet af brikker i den givne `island`. Ved at gange disse tal sammen, finder vi scoren for hver `island`. Dette gøres for hver blob `island`, disse lægges sammen og derved får vi den endelige score.

3. Evaluering og resultater

Til at lave vores testdata har vi brugt information fra de første 59 billeder, og vil derfor bruge de 15 billeder 60-74 til at evaluere vores program. Testen forløb ved at køre programmet, og beregne scoren for hvert af de sidste billeder. Hvilket gav følgende resultater:

Plade nr:	Antal point (Ground truth)	Antal point talt af programmet:
60	44	53
61	64	181
62	36	48
63	38	27
64	66	94
65	80	117
66	124	150
67	99	93
68	66	72
69	124	150
70	99	87
71	66	46
72	80	144
73	124	108
74	99	37

Som det kan ses på tabellen har programmet ikke ramt det rigtige antal point på nogle af spillepladerne (som f.eks på figur 4 hvor (1,0) ikke bliver registreret korrekt og den derfor splitter et territorium op).

Figur 4: Gameboard nr. 74

Derfor vil vi se på hvor mange tiles den har fået rigtig, da en enkelt krone, eller tile fra eller til kan have forholdsvis store udsving i antal af point man kan få på en plade.

Tiletype	Antal tiles i alt	Antal tiles talt af programmet	Falske positiver	Falske Negativer
Desert 0 kroner	78	66	0	12
Desert 1 krone	19	28	9	0
Forrest 0 kroner	63	69	10	4
Forrest 1 krone	23	19	7	11
Water 0 kroner	45	45	3	3
Water 1 krone	24	20	0	4
Wasteland 0 kroner	23	24	12	11

Wasteland 1 krone	7	14	11	4
Wasteland 2 kroner	8	13	9	4
Grassland 0 kroner	32	8	0	24
Grassland 1 krone	6	30	26	2
Grassland 2 kroner	6	4	2	4
Mine 0 kroner	3	5	2	0
Mine 1 krone	3	1	1	3
Mine 2 kroner	9	5	1	5
Mine 3 kroner	3	4	2	1

Som man kan se af tabellen, er programmet rimeligt godt til at kende Deserts (Precision, Recall D0 = 100% , 85 % D1 = 68%, 100%) og Water (Precision, Recall W0 = 93% , 93% , W1 = 100% , 83%) tiles, dette skyldes højst sandsynligt at de har nogle kraftige farver, der skiller sig ud fra de andre. Desuden ændre den ting i midten der indikerer at de har en krone også væsentligt på tilens gennemsnit farve.
Dem som ikke klarede sig så godt er grassland (Precision, Recall G0 = 100% , 25 % G1 = 13%, 67% G2 = 50% , 33%), og wasteland (Precision, Recall WL0 = 50% , 52% WL1 = 21%, 43% WL2 = 31% , 50%). Grunden til at de har klaret sig dårligt, er at deres farver i 3D colorsspace ligger meget tæt på hinanden, og der derfor ikke skal meget forstyrrende fra kanter eller lignende, for at man lige pludselig er ovre i en anden farve tile. en anden grund til det gik så dårligt, er at disse også i høj grad blev forvekslet med hinanden, da wastelandtilesne ikke ændrede særligt meget farve når antallet af kroner skiftede.

4. Konklusion

Programmet kan i dets nuværende stadie ikke bruges til at vurdere antal point ud fra en given spillerplade, da den i 0% af tilfældene ramte rigtigt. Noget der ville gøre programmet mere robust ville være en eller anden form for object genkendelse af midten af hvert tile, da der her er en feature som kan bruges til at beskrive antallet af kroner. Vi har prøvet at bruge gaussian blur på billedet, og trække det fra for at få fat i featuren, men uden held.