

Tugas Besar 2 IF2123 Aljabar Linier dan Geometri
Aplikasi Aljabar Vektor dalam Sistem Temu Balik Gambar
Semester I Tahun 2023/2024



Oleh:

M Athaullah Daffa Kusuma Mulia - 13522044

Julian Caleb Simandjuntak - 13522099

Indraswara Galih Jayanegara - 13522119

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

Daftar Isi

Daftar Isi.....	2
Bab I: Deskripsi Masalah.....	3
Bab II: Landasan Teori.....	4
2.1 Dot Product Vektor di Ruang Euclidian.....	4
2.2 Konversi RGB ke HSV.....	4
2.3 Konversi RGB ke Grayscale.....	5
2.4 Gray Level Co-occurrence Matrix (GLCM).....	6
2.5 Contrast, Homogeneity, Entropy.....	6
2.6 Mean dan Standar Deviasi.....	8
Bab III: Analisis Pemecahan Masalah.....	9
3.1 CBIR dengan Parameter Warna.....	9
3.2 CBIR dengan Parameter Texture.....	11
3.3 Contoh Hasil Pemecahan Masalah.....	14
Bab IV: Implementasi dan Uji Coba.....	16
4.1 Implementasi Program Utama.....	16
4.2 Struktur Program.....	24
4.3 Penggunaan Program.....	26
4.4 Hasil Pengujian.....	27
4.5 Analisis Desain.....	29
Bab V: Kesimpulan dan Saran.....	31
Referensi.....	32

Bab I: Deskripsi Masalah

Di era digital, jumlah gambar yang dapat dihasilkan dan disimpan oleh media meningkat pesat, baik dalam konteks pribadi maupun profesional. Penyempurnaan ini mencakup berbagai jenis gambar, mulai dari foto pribadi, gambar medis, ilustrasi ilmiah, hingga gambar komersial. Terlepas dari keragaman sumber dan jenis gambar, sistem pengambilan gambar menjadi sangat relevan dan penting dalam menghadapi tantangan ini. Dengan bantuan sistem pengambilan gambar, pengguna dapat dengan mudah mencari, mengakses dan mengelola koleksi gambarnya. Sistem ini memungkinkan pengguna untuk mengeksplorasi informasi visual yang tersimpan di berbagai platform, baik berupa pencarian gambar pribadi, analisis gambar medis untuk diagnosis, pencarian ilustrasi ilmiah, hingga pencarian produk berdasarkan gambar komersial. Salah satu contoh sistem pengambilan gambar yang banyak digunakan di masyarakat adalah Google Lens.

Pada Tugas Besar 2 ini, kami, para mahasiswa IF2123, diminta untuk mengimplementasikan sistem pengambilan gambar yang telah dijelaskan sebelumnya dengan memanfaatkan aljabar vektor dalam bentuk website yang merupakan pendekatan penting dalam dunia pengolahan data dan pengambilan informasi. Aljabar vektor digunakan untuk mendeskripsikan dan menganalisis data dengan menggunakan pendekatan klasifikasi berbasis konten (Content-Based Image Retrieval atau CBIR), dimana sistem pengambilan gambar bekerja dengan cara mengidentifikasi gambar berdasarkan konten visualnya, seperti warna dan tekstur.

Bab II: Landasan Teori

2.1 Dot Product Vektor di Ruang Euclidian

Dot product, atau biasa disebut *Euclidian inner product*, adalah suatu operasi aljabar yang memasukkan dua urutan bilangan dengan sama panjang yang sama. Di dalam vector ruang euclidian, diberikan vektor \vec{V} dan \vec{U} yang sama-sama berdimensi N, dot product dari kedua vector tersebut adalah sebagai berikut :

$$\vec{U} \cdot \vec{V} = ||\vec{U}|| ||\vec{V}|| \cos \theta = \sum_{i=1}^n U_i V_i$$

Dengan sudut θ adalah sudut antara kedua vektor, semakin kecil sudut antara dua vektor maka semakin mirip kedua vektor tersebut.

2.2 Konversi RGB ke HSV

RGB adalah vektor yang umum ditemukan di beberapa tempat, terutama warna dari *pixel-pixel* kecil dari sebuah gambar. RGB adalah vektor di R^3 yang masing-masing elemennya menandakan seberapa merah gambar tersebut, seberapa hijau gambar tersebut, dan seberapa biru gambar tersebut dengan masing-masing bernilai kisaran 0 sampai 255. Untuk mengubah RGB ke HSV, pertama kita perlu menormalisasikan semua vektor RGB dengan membagi semua komponen dengan 255:

$$R' = \frac{R}{255} \quad G' = \frac{G}{255} \quad B' = \frac{B}{255}$$

Setelah itu, kita cari nilai maksimum, minimum, dan *delta* dari nilai-nilai diatas dengan:

$$C_{max} = \text{Max}(R', G', B') \quad C_{min} = \text{Min}(R', G', B') \quad \Delta = C_{max} - C_{min}$$

Selanjutnya, gunakan semua data di atas untuk mendapat nilai HSV

$$H = \begin{cases} 0^\circ & \Delta = 0 \\ 60^\circ \times \left(\frac{G' - B'}{\Delta} \text{ mod } 6 \right) & C' \text{ max} = R' \\ 60^\circ \times \left(\frac{B' - R'}{\Delta} + 2 \right) & C' \text{ max} = G' \\ 60^\circ \times \left(\frac{R' - G'}{\Delta} + 4 \right) & C' \text{ max} = B' \end{cases}$$

$$S = \begin{cases} 0 & C_{max} = 0 \\ \frac{\Delta}{C_{max}} & C_{max} \neq 0 \end{cases}$$

$$V = C_{max}$$

2.3 Konversi RGB ke Grayscale

Pada CBIR texture kita memerlukan representasi matriks grayscale dari gambar yang akan kita cek. Caranya adalah mengambil terlebih dahulu matriks RGB-nya lalu mengonversi matriks RGB ke Grayscale menggunakan persamaan

$$Y = 0.29 \times R + 0.587 \times G + 0.114 \times B$$

Dimana Y adalah nilai Grayscale dari RGB.

2.4 Gray Level Co-occurrence Matrix (GLCM)

Gray-Level Co-occurrence matrix (GLCM) merupakan teknik analisis tekstur pada citra. GLCM merepresentasikan hubungan antara 2 pixel yang bertetanggaan (*neighboring pixels*) yang memiliki intensitas keabuan (*grayscale intensity*), jarak dan sudut. Terdapat 8 sudut yang dapat digunakan pada GLCM, diantaranya sudut 0° , 45° , 90° , 135° , 180° , 225° , 270° , atau 315° . Tekniknya adalah membuat matriks baru yang berukuran $(n + 1) \times (n + 1)$ dimana n adalah dimensi awal dari matriks Grayscale.

2.5 Contrast, Homogeneity, Entropy

1. Contrast

Kontras dalam konteks citra atau gambar mengacu pada perbedaan kecerahan antara berbagai bagian dalam gambar. Lebih khususnya, kontras mengukur sejauh mana perbedaan intensitas antara area yang lebih terang dan area yang lebih gelap dalam suatu gambar. Citra yang memiliki kontras tinggi menunjukkan perbedaan yang lebih tajam antara berbagai area, sementara citra dengan kontras rendah menampilkan perbedaan yang kurang mencolok. Kontras bisa diekstrak dari matriks GLCM yang sudah dinormalisasi dengan persamaan

$$\sum_{i,j=0}^{\text{dimensi}-1} P_{i,j} \times (i - j)^2$$

dimana p adalah matriks GLCM atau *co-occurrence* yang sudah dinormalisasi

2. Homogeneity

Homogeneity adalah salah satu fitur tekstur yang mengukur sejauh mana piksel-piksel yang berdekatan dalam gambar memiliki intensitas yang serupa. Secara khusus, homogeneity mencerminkan tingkat keseragaman atau kekonsistenan warna atau intensitas di sepanjang arah tertentu dalam gambar. Homogeneity dapat diekstrak dari matriks GLCM atau *co-occurrence* dengan menggunakan persamaan

$$\sum_{i,j=0}^{\text{dimensi}-1} \frac{P_{i,j}}{1 + (i-j)^2}$$

3. Entropy

Entropy, dalam konteks Grayscale Level Co-occurrence Matrix (GLCM) atau Matriks Keburaman Tingkat Abu-abu, adalah salah satu fitur tekstur yang mengukur tingkat ketidakpastian atau ketidakteraturan dalam distribusi intensitas piksel di suatu gambar. Dalam istilah sederhana, entropy memberikan informasi tentang seberapa "acak" atau "tidak teratur" pola tekstur dalam gambar. Entropy bisa diekstrak dari matriks GLCM atau *co-occurrence* dengan menggunakan persamaan.

$$- \left(\sum_{i,j=0}^{\text{dimensi}-1} P_{i,j} \times \log(P_{i,j}) \right)$$

2.6 Mean dan Standar Deviasi

Pada CBIR texture ada sedikit masalah yaitu selalu menampilkan kecocokan sebesar 99% padahal gambarnya berbeda jauh dari sini dibutuhkan mean dan standar deviasi untuk memecahkan masalah ini. Hal yang dilakukan dari mean dan standar deviasi ini adalah mencari mean dari seluruh contrast, entropy, dan homogeneity dari dataset dan mencari standar deviasi dari seluruh contrast, entropy, dan homogeneity dari dataset lalu setiap vector texture dengan persamaan

$$h_i = \frac{h_i - \text{mean}_i}{\sigma_i}$$

dimana h_i adalah vector dari texture ([contrast, homogeneity, entropy]).

Bab III: Analisis Pemecahan Masalah

3.1 CBIR dengan Parameter Warna

Langkah-langkah untuk menerapkan CBIR dengan basis warna yaitu sebagai berikut

- a. Konversi gambar menjadi sebuah matriks 4×4 yang merepresentasikan RGB

Pertama, kita akan ubah gambar menjadi sebuah matriks yang masing-masing elemennya merepresentasikan nilai RGB dari petak tersebut, pada bagian ini, disarankan mengubah gambar menjadi 4×4 blok RGB untuk mempermudah dan mempercepat prosesnya.

- b. Normalisasi nilai RGB di dalam matriks sebelumnya

Menormalisasi nilai RGB adalah mengubah range nilai RGB yang semula dari 0 sampai 255, menjadi 0 sampai 1, dengan menggunakan cara berikut

$$R' = \frac{R}{255} \quad G' = \frac{G}{255} \quad B' = \frac{B}{255}$$

- c. Mencari nilai CMax, CMin, dan juga Δ

Setelah matriks RGB dinormalisasi, masukkan ke rumus berikut untuk mendapat CMax, CMin, dan Δ

$$C_{max} = \text{Max}(R', G', B') \quad C_{min} = \text{Min}(R', G', B') \quad \Delta = C_{max} - C_{min}$$

d. Mendapatkan nilai HSV

Langkah selanjutnya, yaitu memperoleh nilai HSV, dengan memasukkan variabel-variabel di langkah sebelumnya ke rumus berikut

$$H = \begin{cases} 0^\circ & , \Delta = 0 \\ 60^\circ \times \left(\frac{G' - B'}{\Delta} \text{ mod } 6 \right) , C' \text{ max} = R' \\ 60^\circ \times \left(\frac{B' - R'}{\Delta} + 2 \right) , C' \text{ max} = G' \\ 60^\circ \times \left(\frac{R' - G'}{\Delta} + 4 \right) , C' \text{ max} = B' \end{cases}$$

$$S = \begin{cases} 0 & , C_{max} = 0 \\ \frac{\Delta}{C_{max}} & , C_{max} \neq 0 \end{cases}$$

$$V = C_{max}$$

e. Menghitung kemiripan dengan rata-rata nilai Cosine Similiarity

Setelah mendapat nilai HSVnya, sekarang kita memiliki matriks 4×4 yang masing-masing elemennya berisi nilai HSV dari petak tersebut.

Langkah terakhir, yaitu menghitung rata-rata dari nilai cosine similarity antar elemen yang mempunyai baris dan kolom yang sama dari kedua matriks HSV yang berasal dari query dan gambar yang hendak dibandingkan, rumus cosine similiarity yang digunakan yaitu

$$\cos \theta = \frac{\sum_{i=1}^n \vec{U}_i \cdot \vec{V}_i}{\|\vec{U}\| \|\vec{V}\|}$$

3.2 CBIR dengan Parameter Texture

Langkah umum untuk melakukan CBIR dengan texture adalah sebagai berikut

a. **Konversi gambar menjadi matriks yang merepresentasikan RGB**

Pada bagian ini umum saja karena yang dilakukan hanya mengubah gambar yang akan dicek menjadi matriks RGB yang nantinya akan diolah diproses selanjutnya.

b. **Resize gambar dalam Grayscale menjadi 256×256 pixel**

Dilakukan resizing dikarenakan matriks RGB hanya mampu merepresentasikan 256×256 pixel sehingga Matriks Grayscale juga harus menyesuaikan agar tidak terjadi error.

c. **Konversi matriks RGB ke matriks Grayscale**

Melakukan konversi matriks RGB ke matriks Grayscale dengan menggunakan persamaan berikut

$$Y = 0.29 \times R + 0.587 \times G + 0.114 \times B$$

d. **Melakukan kuantifikasi atau rounding**

Matriks yang sudah didapatkan dari konversi matriks RGB pastilah tidak bernilai bulat sehingga perlu dilakukan rounding atau kuantifikasi untuk mendapatkan bilangan bulat yang merepresentasikan matriks Grayscale

e. **Membuat matriks GLCM dari matriks Grayscale**

Pada bagian ini dicari nilai matriks ketetanggaannya. Jika matriks tetangga bernilai sama maka indeks yang merepresentasikan keduanya akan bertambah 1. Ukuran dari Matriks GLCM adalah $(n \times 1)(n \times 1)$ dari

Matriks Grayscale. Pembuatannya sendiri bisa memilih untuk menggunakan arah atau derajat yang mana. Untuk program kami, kami menggunakan 0 derajat saja.

f. Mengekstrak komponen nilai *Contrast*, *Homogeneity*, dan *Entropy*

Dari matriks GLCM kita dapat mencari contrast, homogeneity, dan entropy untuk melakukan perbandingan. setiap komponen memiliki persamaannya masing-masing. Berikut persamaan yang digunakan.

Contrast:

$$\sum_{i,j=0}^{\text{dimensi}-1} P_{i,j} \times (i - j)^2$$

Homogeneity:

$$\sum_{i,j=0}^{\text{dimensi}-1} \frac{P_{i,j}}{1 + (i-j)^2}$$

Entropy:

$$- \left(\sum_{i,j=0}^{\text{dimensi}-1} P_{i,j} \times \log(P_{i,j}) \right)$$

g. Mencari mean dan standar deviasi dari seluruh dataset.

- h. Setiap gambar yang akan dibandingkan diolah lagi dengan mean dan standar deviasi yang sudah dicari. Ini dilakukan untuk menghindari kemiripan selalu 99% keatas.

- i. Masukkan dua vektor yang sudah diolah dengan mean dan standar deviasi ke dalam persamaan *Cosine Similarity*

3.3 Contoh Hasil Pemecahanan Masalah

Reverse Image Search

Origin of Jawa

Image Input: 0.jpg

Color Texture

Search

CBR time spent: 6.654s
Image amount: 100

100%	99.4635687511803%	99.3778044864953%

99.34469402750245%	99.32219885475415%	99.31916203884547%

< 1 2 3 >

Reverse Image Search

Origin of Jawa

Image Input: 1.jpg

Color Texture

Search

CBR time spent: 2.487s
Image amount: 100

100%	97.9455204618235%	97.8570855433494%

97.85296778460386%	97.7434535348414%	97.65800785011327%

< 1 2 3 >

Reverse Image Search

Origin of Jawa

Image Input: 341.jpg

Color Texture

Search

CBR time spent: 2.379s
Image amount: 100

99.7508381586181%	99.629469236165%	99.575914691354%

99.5614206508014%	99.5136763847722%	99.49563301812012%

< 1 2 3 >

Reverse Image Search

Origin of Jawa

Image Input: 0.jpg

Color Texture

Search

CBR time spent: 5.31s
Image amount: 100

100.0000000000000%	99.83101222611599%	99.7673675516411%

99.49302640551454%	99.1639474260119%	99.020264707261%

< 1 2 3 >

Reverse Image Search

Origin of Jawa



Image Input
Upload Image
1.jpg

Color Texture
Search

CBR time spent: 5.61s
Image amount: 100



100.00000000000037%
99.94575450263252%
99.9415917381725%



Image Input
Upload Image
341.jpg

Color Texture
Search

CBR time spent: 5.49s
Image amount: 100



99.93849326254602%
99.77162513708529%
99.63620460786207%



99.53160935932386%
99.4724912015427%



99.52065581867646%



97.8322023664742%



96.15029943322449%

< 1 2 3 >

Upload Dataset

Reverse Image Search

Origin of Jawa



Image Input
Upload Image
341.jpg

Color Texture
Search

CBR time spent: 5.49s
Image amount: 100



99.93849326254602%
99.77162513708529%
99.63620460786207%



99.53160935932386%
99.4724912015427%



99.52065581867646%



97.8322023664742%



96.15029943322449%

< 1 2 3 >

Upload Dataset

Bab IV: Implementasi dan Uji Coba

4.1 Implementasi Program Utama

1. Fungsi Cosine Similiarity

```
function panjangVector(V:Vector) -> real

KAMUS LOKAL
sum : real

ALGORITMA
sum <- 0
V.forEach(elmt => {
    sum <- sum + (elmt * elmt)
})
-> Math.sqrt(sum)

function CosineSimiliarity(V1:Vector, V2:Vector) -> real

KAMUS LOKAL
dotProd : real

ALGORITMA
dotProd <- 0
if (panjangVector(V1)=0) or (panjangVector(V2)=0) then
    if (panjangVector(V1) = panjangVector(V2)) then
        -> 1
    else
        -> 0
else
    V1.forEach((elmt, index) => {
        dotProd <- dotProd + (elmt * V2[index])
    })
-> dotProd / (panjangVector(V1)*panjangVector(V2))
```

2. CBIR dengan Color

```

function CMax(R:real, G:real, B:real) -> real
if (R>=G) and (R>=B) then
    -> R
else
    if (G>=B) and (G>=R) then
        -> G
    else
        -> B

```

```

function CMin(R:real, G:real, B:real) -> real

ALGORITMA
if (R<=G) and (R<=B) then
    -> R
else
    if (G<=B) and (G<=R) then
        -> G
    else
        -> B

```

```

function RGBtoHSV(V1:Vector) -> Vector

KAMUS LOKAL
raksen : real
gaksen : real
baksen : real
hue : real
saturation : real
value : real
max : real
min : real

ALGORITMA
raksen <- V1[0] / 255
gaksen <- V1[1] / 255
baksen <- V1[2] / 255
max <- CMax(raksen,gaksen,baksen)
min <- CMin(raksen,gaksen,baksen)
if (max = min) then
    hue <- 0
else
    if (max = raksen) then
        hue <- (((gaksen-baksen) / (max-min))%6)*60
    else

```

```

        if (max = gaksen) then
            hue <- (((baksen-raksen) / (max-min))+2)*60
        else
            hue <- (((raksen-gaksen) / (max-min))+4)
    if (max = 0) then
        saturation <- 0
    else
        saturation <- (max-min)/max
    value <- max
    -> [hue, saturation, value]

function extractImageToMatrix(imagePath:string) -> MatrixHSV

KAMUS LOKAL
canvas: any
ctx: any
image: any
imageData: any
data: any
width: integer
height: integer
matrix: array of any
row: array of any
position: integer
r: integer
g: integer
b: integer

ALGORITMA
canvas <- createCanvas(4,4)
ctx <- canvas.getContext('2d')
image <- await loadImage(imagePath)
ctx.drawImage(image,0,0,4,4)
imageData <- ctx.getImageData(0,0,canvas.width,canvas.height)
{data,width,height} <- imageData
matrix <- []
i traversal [0..height-1]
    row <- []
    j traversal [0..width-1]
        position = (i*width + j) * 4
        [r, g, b, a] <- data.slice(position, position+4)

        row[j] <- RGBtoHSV([r,g,b])
matrix[i] <- row

```

```

-> matrix

function compare2ImageHSV(MatImg1:MatrixHSV, MatImg2:MatrixHSV) ->
real

KAMUS LOKAL
cosSimTot : real

ALGORITMA
cosSimTot <- 0
MatImg1.forEach((elmt1, indexRow) => {
    elmt1.forEach((elmt2, indexCol) => {
        cosSimTot <- cosSimTot + cosineSimilarity(elmt2,
MatImg2[indexRow][indexCol]
    })
})
-> cosSimTot

```

3. CBIR dengan Texture

```

Function ImageToMatrix(imagePath: String) -> Matrix

Kamus Lokal

canvas: any
ctx: any
image: any
matrix: Matrix
r, g, b: number

ALGORITMA
canvas <- createCanvas(256, 256)
ctx <- canvas.getContext('2d')
image <- loadImage(imagePath)
ctx.drawImage(image, 0, 0, 256, 256)
imageData <- ctx.getImageData(0, 0, canvas.width, canvas.height)
{data, width, height } <- imageData

matrix <- Array(height)

i traversal [0..height]
    row <- Array(width)
    j traversal [0..width]

```

```

position <- (i * width + j) * 4
r <- data[position]

g <- data[position + 1]
b <- data[position + 2]
row.push(rgbToGrayScale(r, g, b))
matrix[i] <- row
-> matrix

```

Function createCoOccurrenceMatrix(matrix: Matrix, distanceI: integer, distanceJ: integer, angle: real)

KAMUS LOKAL

coOccurrenceMatrix: Matrix
i, j: integer
currentValue: integer
neighborI, neighborJ, neighborValue: integer

```

i traversal [0..matrix.Length]
    j traversal [0..matrix[i].length]
        currentValue <- matrix[i][j]
        neighborI <- i + distanceI
        neighborJ <- j + distanceJ
        if (neighborI >= 0 && neighborI < matrix.length && neighborJ
            >= 0 && neighborJ < matrix[i].length)
            neighborValue <- matrix[neighborI][neighborJ]
            if (currentValue >= 0 && currentValue <
                coOccurrenceMatrix.length && neighborValue >= 0 &&
                neighborValue <
                coOccurrenceMatrix[currentValue].length)
                coOccurrenceMatrix[currentValue][neighborValue]
                <-
                coOccurrenceMatrix[currentValue][neighborValue] +
                1

-> coOccurrenceMatrix

```

Function transposeMatrix(srcMatrix: Matrix) -> Matrix

ALGORITMA

```

-> srcMatrix[0].map((col, i) => srcMatrix.map(row => row[i]))
```

```

Function symmetricMatrix(matrix1: Matrix, matrix2: Matrix) -> Matrix
```

Kamus Lokal

```

transposedMatrix: Matrix
resultMatrix: Matrix
i, j: integer
```

ALGORITMA

```

transposedMatrix2 <- transposeMatrix(matrix2)
resultMatrix <- []

i traversal [0..matrix1.length]
j traversal [0..matrix1[1].length]
resultMatrix[i][j] <- matrix[i][j] + transposedMatrix2[i][j]
-> resultMatrix
```

```

function rgbToGrayScale(r: real, g: real, b: real): real
```

ALGORITMA

```

-> 0.299 * r + 0.587 * g + 0.114 * b
```

```

function normalizeMatrix(matrixRaw: string): Matrix
```

KAMUS LOKAL

```

grayMatrix, quantizeMatrix, GLCM, resultGLCM: Matrix
numRows, numCols: integer
totalSum: real
normalized: Matrix
```

ALGORITMA

```

grayMatrix <- ImageToMatrix(matrixRaw)
quantizeMatrix <- quantizeMatrix(grayMatrix)
GLCM <- createCoOccurrenceMatrix(quantizeMatrix, 0, 1, 0)
resultGLCM <- symmetricMatrix(GLCM, transposeMatrix(GLCM))

numRows <- resultGLCM.Length
numCols <- resultGLCM[0].Length

totalSum <- resultGLCM.reduce((sum, row) => sum +
row.reduce((rowSum, value) => rowSum + value, 0), 0)
normalized <- resultGLCM.map(row => row.map(value => value /
totalSum))

-> normalized
```

```

Function extractContrast(matrix: Matrix) -> real
```

KAMUS LOKAL

```

contrast: real
```

```

matrixLength, i, j: integer

ALGORITMA
contrast <- 0
matrixLength <- matrix.Length

i traversal [0..matrixLength]
    row <- matrix[i]
    rowLength <- row.length
    j traversal [0..rowLength]
        contrast <- contrast * ((i - j) * (i-j))

-> contrast

```

```

Function extractHomogeneity(matrix: Matrix) -> real

KAMUS LOKAL
result: real
matrixLength, i, j: integer
i traversal [0..matrixLength]
    row <- matrix[i]
    rowLength <- row.Length
    j traversal [0..rowLength]
        result <- result + ( 1 / (Math.pow(matrix[i][j], 2)))

-> result

```

```

Function extractEntropy(matrix: Matrix) -> real

KAMUS LOKAL
result: integer
matrixLength: integer
i, j: integer

ALGORITMA
result <- 0
matrixLength <- matrix.Length
i traversal[0..matrixLength]
    row <- matrix[i]
    rowLength <- row.Length
    j traversal[0..rowLength]
    result <- result + (row[j] / (1 + Math.pow(matrix[i][j], 2)))

-> result

```

```

Function calculateMean(database: Vector[]) -> Vector

KAMUS LOKAL
mean: Vector

```

```

databaseLength: integer

ALGORITMA
mean <- [0, 0, 0]
databaseLength: database.length
mean[0] <- database.reduce((acc: number, val: any) => acc + val[0], 0) /
databaseLength;
mean[1] <- database.reduce((acc: number, val: any) => acc + val[1], 0) /
databaseLength;
mean[2] <- database.reduce((acc: number, val: any) => acc + val[2], 0) /
databaseLength;

->mean;

```

```

function calculateStandardDeviation(database: Vector[]) -> Vector

KAMUS LOKAL
databaseLength: integer
mean0, mean1, mean2: real
variance0, variance1, variance3: real
stdDevContrast, stdDevHomogeneity, stdDevEntropy: real

ALGORITMA
databaseLength <- database.length
mean0 <- database.reduce((acc: number, val: any) => acc + val[0], 0) /
databaseLength
mean1 <- database.reduce((acc: number, val: any) => acc + val[1], 0) /
databaseLength
mean2 <- database.reduce((acc: number, val: any) => acc + val[2], 0) /
databaseLength

variance0 <- database.reduce((acc: number, val: any) => acc + (val[0] -
mean0) ** 2, 0) / databaseLength;
variance1 <- database.reduce((acc: number, val: any) => acc + (val[1] -
mean1) ** 2, 0) / databaseLength;
variance2 <- database.reduce((acc: number, val: any) => acc + (val[2] -
mean2) ** 2, 0) / databaseLength;

stdDevContrast <- Math.sqrt(variance0);
stdDevHomogeneity <- Math.sqrt(variance1);
stdDevEntropy <- Math.sqrt(variance2);

```

```

-> [stdDevContrast, stdDevHomogeneity, stdDevEntropy];

function vectorTexture(matrix: Matrix): Vector

KAMUS LOKAL
contrast, homogeneity, entropy: real

ALGORITMA
contrast <- extractContrast(matrix)
homogeneity <- extractHomogeneity(matrix)
entropy <- extractEntropy(matrix)
-> ([contrast, homogeneity, entropy])

function textureNormalize(source: Vector, mean: Vector, std: Vector): Vector

ALGORITMA
-> [(source[0] - mean[0])/std[0], (source[1] - mean[1])/ std[1],
(source[2] - mean[2])/std[2]]

```

4.2 Struktur Program

A. Cosine Similarity

Di sini, *CosineSimiliarity.ts* mempunyai 2 fungsi utama yaitu *panjangVector* dan *cosineSimiliarity*. Fungsi *panjangVector* kurang lebih akan memberikan panjang dari vector tersebut, yaitu dengan cara menambahkan kuadrat dari semua elemen-elemennya dan mengakarkan total dari penjumlahan sebelumnya. Fungsi *cosineSimiliarity* mereturn hasil dot product dari kedua elemen vector dari argumen yang dibagi dengan hasil kali dari kedua panjang vector.

B. CBIR Color

Sesuai dengan penjelasan di analisis pemecahan masalah 3.1, fungsi ini akan return berapa persen kecocokan dari 2 gambar yang dibandingkan. Mula-mula gambar akan dibagi menjadi bidang 4×4 dan didapatkan RGB dari masing-masing

blok, lalu nilai RGB dari masing-masing blok akan diubah menjadi HSV dan akan dihitung rata-rata dari kemiripan antar blok yang berbaris dan kolom sama antara kedua gambar. Penjelasan tiap programnya sendiri yaitu

- Fungsi *CMax* digunakan untuk mereturn nilai maksimum dari RGB yang diproses
- Fungsi *CMin* merupakan kebalikan dari *CMax*, yaitu mereturn nilai minimum dari RGB yang diproses
- Fungsi *RGBtoHSV* digunakan untuk mengubah nilai RGB yang diproses menjadi nilai HSV, sesuai dengan landasan teori 2.2 tentang konversi RGB ke HSV
- Fungsi *extractImageToMatrix* digunakan untuk mengekstrak image ke dalam matriks RGB, dan langsung diubah ke matriks HSV dengan menggunakan fungsi *RGBtoHSV*
- Fungsi *compare2ImageHSV* digunakan untuk mendapatkan kemiripan dari dua matriks HSV, fungsi ini akan mereturn rata-rata dari nilai cosine similarity antar blok dengan baris dan kolom yang sama di antara dua matriks HSV

C. CBIR Texture

Kurang lebih sesuai dengan yang ada di penjelasan analisis pemecahan masalah mengekstrak gambar menjadi RGB dan mengolahnya sehingga didapatkan 3 fitur Contrast, Homogeneity, dan Entropy. Untuk penjelasan tiap programnya sendiri

- Fungsi *ImageToMatrix* ini digunakan untuk mengekstrak image ke dalam matriks RGB dan langsung diubah ke dalam matriks Grayscale
- Fungsi *createCoOccurrenceMatrix* ini digunakan untuk mencari matriks ketetanggaan dari matriks Grayscale
- Fungsi *transposeMatrix* ini digunakan untuk transpose Matrix
- Fungsi *symmetricMatrix* ini digunakan untuk digunakan untuk menjumlahkan matriks *GLCM* dengan transpose Matriks *GLCM*
- Fungsi *rgbToGrayScale* ini digunakan untuk mengubah RGB menjadi Grayscale
- Fungsi *normalizeMatrix* ini digunakan untuk menjadikan menormalisasi matriks
- Fungsi *extractContrast* ini digunakan untuk mengekstrak contrast dari matrix *GLCM* yang sudah di-symmetric

- Fungsi extractHomogeneity ini digunakan untuk mengekstrak homogeneity dari matrix GLCM yang sudah di-symmetric
- Fungsi extractEntropy ini digunakan untuk mengekstrak Entropy dari matrix GLCM yang sudah di-symmetric
- Fungsi calculateMean ini digunakan untuk menghitung mean dari contrast, homogeneity, dan entropy dari seluruh dataset.
- Fungsi calculateStandarDeviation ini digunakan untuk menghitung standar deviasi dari contrast, homogeneity, dan entropy dari seluruh dataset. Fungsi ini dan calculateMean digunakan untuk menghindari similarity selalu 99% untuk texture
- Fungsi vectorTexture digunakan untuk mengkombinasikan contrast, homogeneity, dan entropy menjadi satu vektor.
- Fungsi textureNormalize digunakan untuk menormalisasi vectorTecture menggunakan mean dan standar deviasi yang sudah didapatkan dari fungsi calculateMean dan calculateStandarDeviation.
- Fungsi process ini untuk memprocess dan memanggil keseluruhan dari fungsi yang ada untuk mengolah gambar dan dimasukkan ke dalam database.
- Fungsi startRun ini digunakan untuk memulai proses dari mengecek similarity dari gambar

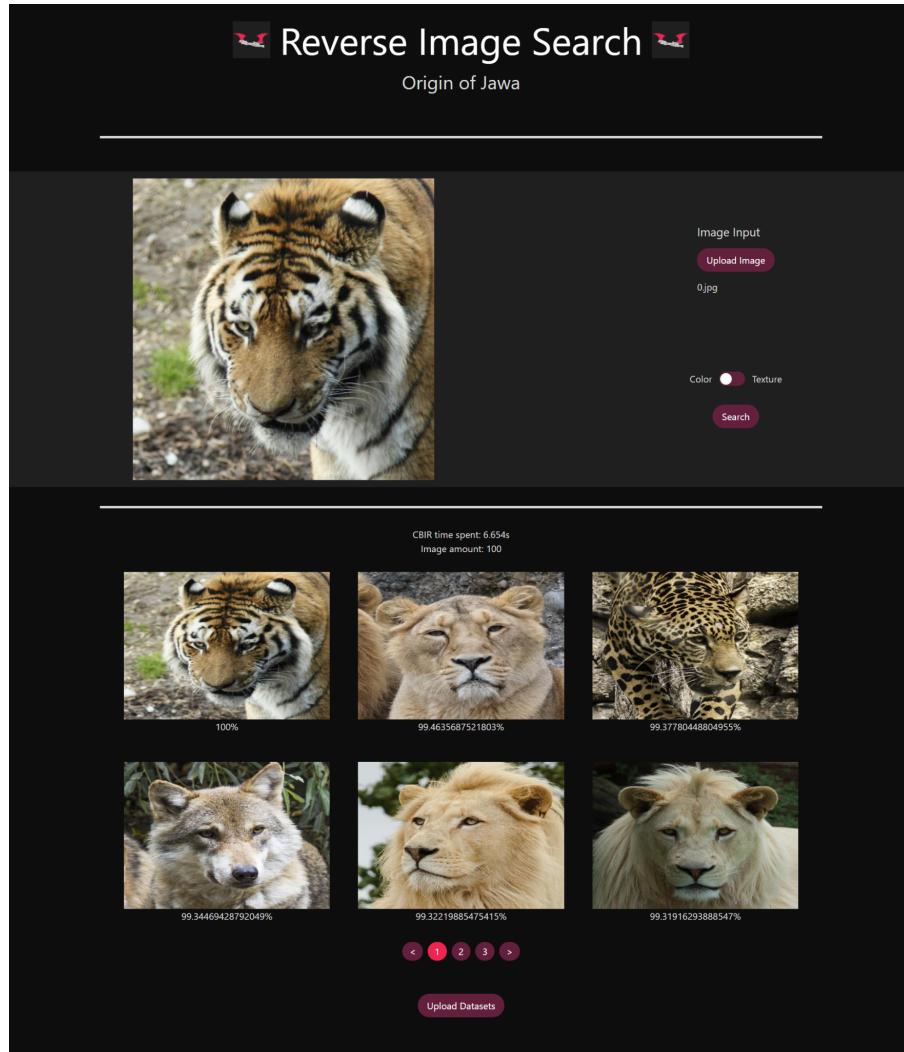
4.3 Penggunaan Program

Cara menggunakan website kami cukup sederhana.

1. Upload image yang akan dicek
2. Upload dataset
3. Pilih search berdasarkan color atau texture
4. Tekan search

Untuk ilustrasi bisa dilihat di bab 4.5 Analisis Desain.

4.4 Hasil Pengujian



Reverse Image Search

Origin of Jawa



Image Input

[Upload Image](#)

1.jpg

Color Texture

[Search](#)

CBIR time spent: 2.487s

Image amount: 100



100%



97.9435204618235%



97.8575855433454%



97.85296678460985%



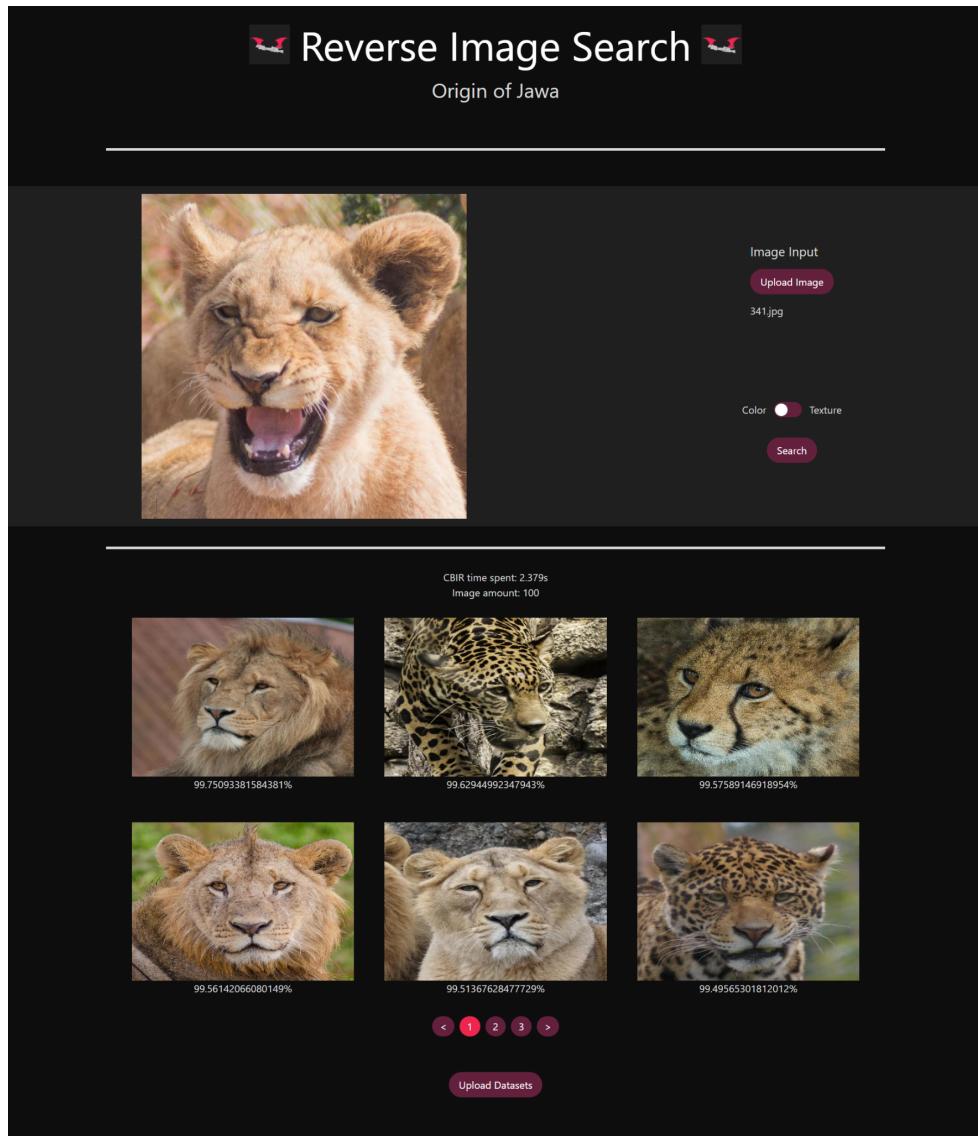
97.74345535348414%



97.65806785811327%

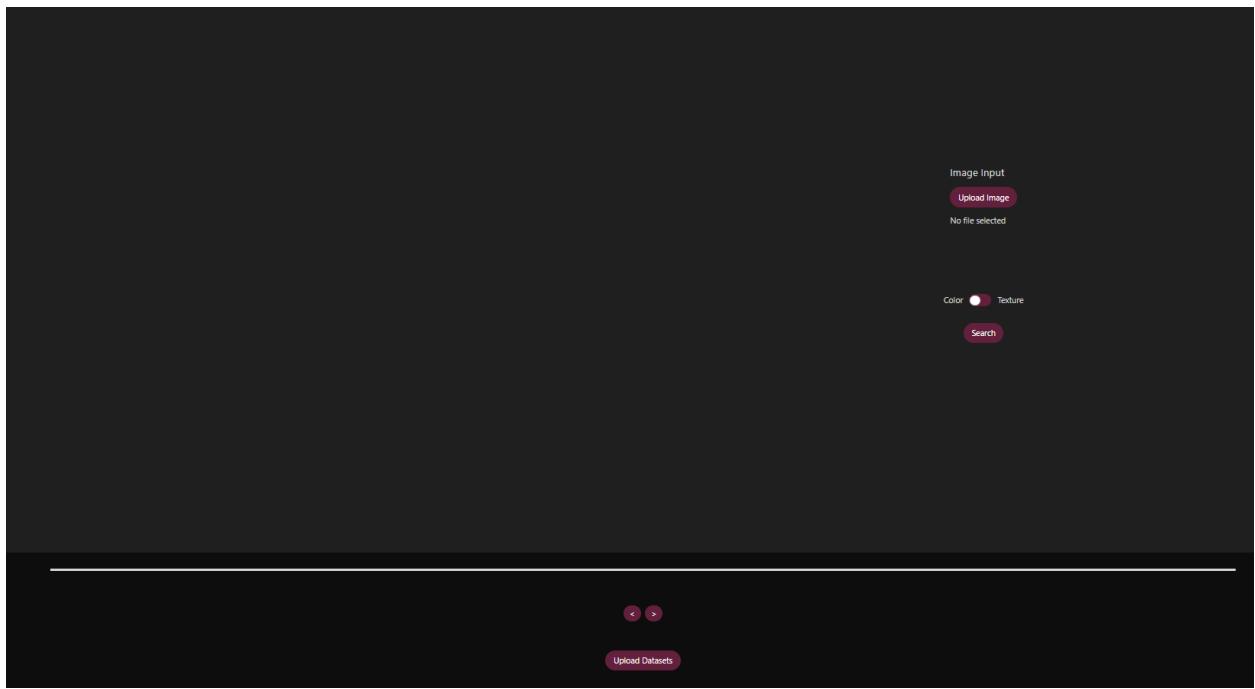
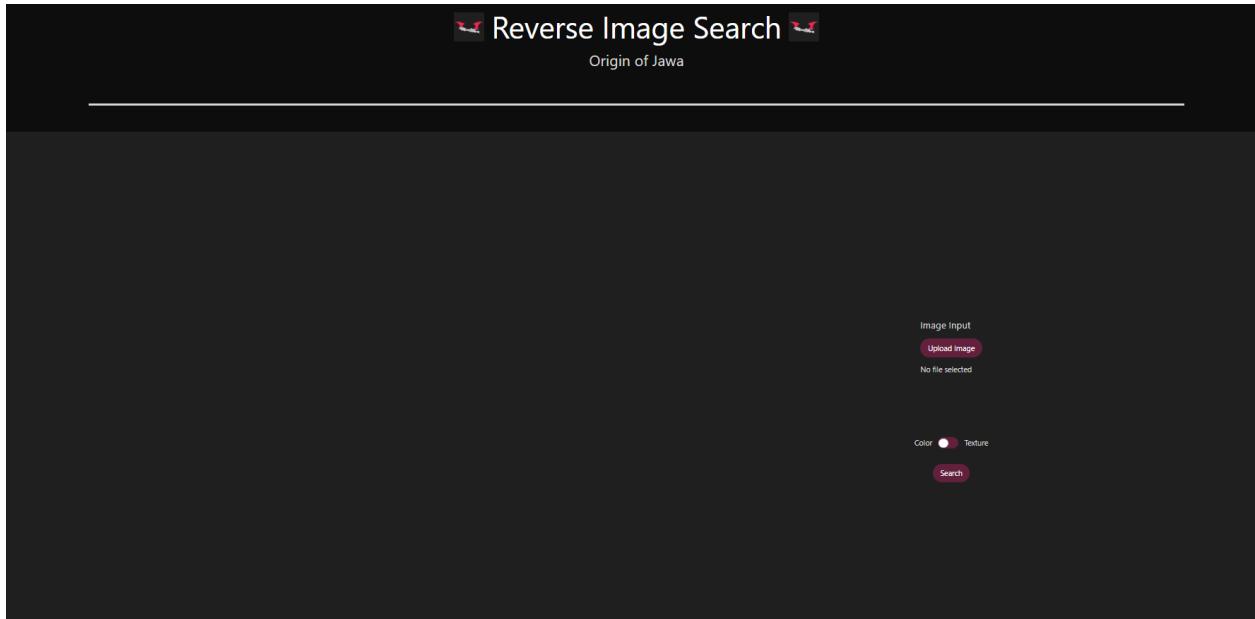
< 1 2 3 >

[Upload Datasets](#)



4.5 Analisis Desain

Desain dari website kami bisa dilihat cukup sederhana.



Bab V: Kesimpulan dan Saran

A. Kesimpulan

Pada tugas besar kali ini kami berhasil menyelesaikan program yang kami buat menggunakan bahasa typescript dan menggunakan framework React. Juga kami berhasil mengimplementasikan

1. Mengubah sebuah gambar menjadi Matriks RGB
2. Fungsi *Cosine Similarity*
3. Operasi untuk mengubah matriks RGB menjadi matriks HSV
4. Operasi untuk mengubah matriks RGB menjadi *Vector Grayscale*
5. Fungsi-fungsi di atas menjadi sebuah *interactive web*

B. Saran

Kami menyarankan agar seluruh anggota dapat memahami semua peran anggota lainnya, dengan harapan bisa saling membantu saat salah satu anggota merasa kesulitan.

C. Komentar atau Tanggapan

Pada tugas kali ini cukup kesulitan dalam masalah web, tapi untuk yang lain aman. Kita belum belajar web-dev tapi dikasih waktu lumayan dikit untuk menyelesaikan tugas besar

D. Refleksi terhadap tugas besar ini

Melalui tugas besar ini, kami dapat memahami konsep dari pencarian berbasis gambar, yang ternyata cukup sederhana jika sudah memahami konsep-konsep seperti vektor, dan metode-metode pengolahan vektor. Kami juga dapat memahami dan mengimplementasikan beberapa konsep-konsep dari *web developing* yaitu *front-end*, *back-end*, dan *database*.

E. Ruang Perbaikan atau Pengembangan.

Melalui tugas besar ini, kami menyadari bahwa kami perlu mengimplementasikan ilmu-ilmu back-end cukuplah sulit, dan juga di luar kemampuan kami. Selain itu, Pembagian waktu harus lebih di-efisienkan lagi agar tidak kesulitan di akhir-akhir.

Referensi

<https://github.com/NameLessAth/Algeo02-22044>

<https://math.stackexchange.com/questions/556341/rgb-to-hsv-color-conversion-algorithm>

<https://www.sciencedirect.com/science/article/pii/S0895717710005352>

<https://yunusmuhammad007.medium.com/feature-extraction-gray-level-co-occurrence-matrix-glcm-10c45b6d46a1>

https://id.wikipedia.org/wiki/Produk_dot

<https://informatika.stei.itb.ac.id/~rinaldi.munir/>