Organizador de fotos

Juan de Dios Martínez
Instituto Tecnológico de Costa Rica
2016206482

Nicolás Feoli Chacón
Instituto Tecnológico de Costa Rica
2016

Melisa Cordero Arias
Instituto Tecnológico de Costa Rica
2016126133

June 2, 2017

Abstract

Este documento presenta el análisis de ciertos algoritmos utilizados para el Local Sensitive Hashing (LSH), Local Binary Patterns (LBP) y Pixeles y para lograr llegar a un número hash y con esto lograr ordenar o clasificar la imagen dependiendo de su similitud al resto de imágenes.

1 Introducción

Esta aplicacion fue diseñada con el fin de facilitarle a las personas el hecho de tener que organizar sus fotos cada vez que desean ordenar su teléfono. Este app al seleccionarse una foto desde la cámara o la galería, le hace posible al usuario organizarla en 2 versiones, por medio de sus pixeles, o por medio del conocido algoritmo LBP. Entre estas 2 versiones, es preferible y más óptimo el LBP ya que su funcionamiento es realmente muy cercano a lo deseado, organizando fotos que son similares y distribuyendo las que no, por otro lado, el algoritmo que utiliza los pixeles tiene un funcionamiento bueno, pero no es el mejor. Ambos algoritmos utilizan un método de Local Sensitive Hashing(LSH), el que se encarga de maximizar las colisiones entre imágenes y con ello saber que tan similares son en realidad.

2 Pixeles

2.1 Descripción

Este metodo de ubicación de imagenes segun su similitud es uno de los mas simples, por el hecho de que es sencillo de programar y no es tan necesario que sea preciso o exacto. Para el desarrollo de este algoritmo es necesario tener el método que realice el producto punto, y K planos aleatorios constantes para todas las imágenes para poder llegar a un resultado mas preciso gracias al algoritmo de LSH. Al tener la imagen se utiliza un metodo para poder obtener un vector con los pixeles empaquetados de la imagen, para luego realizar el producto punto entre ese vector y el vector V_1, luego entre el y V_2 hasta realizarlo con V_k y con esto obtener el numero hash de K digitos. Luego se pregunta si el hash ya existe para incluir esta imagen en el bucket correspondiente y si no este se crea.

```
1
2  public void pixeles(Bitmap bitmap){
3          System.out.println(bitmap);
4
5          bitmap.getPixels(vectorImagen
       ,0,256,0,0,256,256);
6          for(int i=0;i<65536;i++){
7              vectorImagen[i] = Math.abs
       (vectorImagen[i]%256);
8          }
9          //AIUDA
10         long hash = productoPunto(
       vectorImagen, "pixels");
11         Bucket b = sing.getBucketPix(
       hash);
12         if(b==null){
13             b = new Bucket(Long.
       toString(hash),hash);
14             sing.insertarBucketPix(b);
15         }
16         b.agregarImagen(bitmap);
17     }
18 }
19
20
```

2.2 Analisis O grande

El algoritmo de analisis de las imágenes por pixeles tiene una O grande de $O(65536 * m)$, pues se ejecuta el algoritmo LSH directamente al vector de la imagen, comparando cada pixel de la imagen con cada pixel de cada hiperplano. En este caso, m es el número de hiperplanos con el que se está trabajando en el momento.

3 LBP

3.1 Descripción

El algoritmo LBP se basa en la generación de un histograma que permite comparar la frecuencia con que ciertos patrones aparecen en un vector. Para ello se compara cada coeficiente del vector con sus vecinos, si el coeficiente que se esta analizando es mayor que el vecino, se coloca un 0 en la posición correspondiente del hash temporal en caso contrario se coloca un 1. Luego se le sumará 1 a la posición [hash] del histograma. Esto se realiza para cada entrada del vector.

```
1
2     public void LBP(Bitmap bitmap){
3         int[] histograma = new int
      [256];
4         for(int i=0; i<256; i++)
      histograma[i] = 0;
5         System.out.println(bitmap);
6         bitmap.getPixels(vectorImagen
      ,0,256,0,0,256,256);
7
8         for(int i=0;i<65536;i++)
9             vectorImagen[i] = Math.abs
      (vectorImagen[i]%256);
10        int hashPixel;
11        for(int i=0; i<65536;i++){
12            hashPixel = 0;
13            if(i<256){//si esta en la
      fila de arriba
14                if(i == 0){
15                    //esta en la
      esquina superior izq
16                    if(vectorImagen[i
      + 1] < vectorImagen[i]) hashPixel
      += 1; //pixel de derecha
17                    hashPixel *= 2; //
      shift left 1
18                    if(vectorImagen[i
      +257] < vectorImagen[i]) hashPixel
       += 1; // pixel de esquina
      inferior derecha
19                    hashPixel *= 2;
20                    if(vectorImagen[i
      +256] < vectorImagen[i]) hashPixel
       += 1; //pixel de abajo
21                    hashPixel *= 4; //
      como esta en la esquina, no se
      revisan los dos ultimos.
22                } else if(i == 255){
      //esta en la esquina superior
      derecha
23                    if(vectorImagen[i
      +256] < vectorImagen[i]) hashPixel
       += 1; //pixel de abajo
24                    hashPixel *= 2; //
      shift left 1
25                    if(vectorImagen[i
      +255] < vectorImagen[i]) hashPixel
       += 1;//pixel esquina inf
      izquierda
26                    hashPixel *= 2;
27                    if(vectorImagen[i
      - 1] < vectorImagen[i]) hashPixel
      += 1; //pixel izquierda
28                } else{
29                    if(vectorImagen[i
      + 1] < vectorImagen[i]) hashPixel
      += 1; //pixel de derecha
30                    hashPixel *= 2; //
      shift left 1
31                    if(vectorImagen[i
      +257] < vectorImagen[i]) hashPixel
       += 1; // pixel de esquina
      inferior derecha
32                    hashPixel *= 2;
33                    if(vectorImagen[i
      +256] < vectorImagen[i]) hashPixel
       += 1; //pixel de abajo
34                    hashPixel *= 2;
35                    if(vectorImagen[i
      +255] < vectorImagen[i]) hashPixel
       += 1; //pixel esquina inf
      izquierda
36                    hashPixel *= 2;
37                    if(vectorImagen[i
      - 1] < vectorImagen[i]) hashPixel
      += 1; //pixel izquierda
38                }
39            }else if(i >= 65536-256){
      //si esta en la fila de abajo
40                if(i == 0){//esta en
      la esquina inferior izquierda
41                    if(vectorImagen[i
      -256] < vectorImagen[i]) hashPixel
       += 1; //pixel arriba
42                    hashPixel *= 2;
43                    if(vectorImagen[i
      -257] < vectorImagen[i]) hashPixel
       += 1; //pixel sup derecha
44                    hashPixel *= 2;
45                    if(vectorImagen[i
```

```
    + 1] < vectorImagen[i]) hashPixel
    += 1; //pixel de derecha
                    hashPixel *= 16;
    //shift left 4, pues no se revisan
     los ultimos
                }else if(i == 65535){
    //esta en la esquina inferior
    derecha
                    if(vectorImagen[i
    -257] < vectorImagen[i]) hashPixel
     += 1; //pixel superior izquierda
                    hashPixel *= 2;
                    if(vectorImagen[i
    -256] < vectorImagen[i]) hashPixel
     += 1; //pixel arriba
                    hashPixel *= 64;
    //shift left 6
                    if(vectorImagen[i
    - 1] < vectorImagen[i]) hashPixel
    += 1;//pixel izquierda
                }else{
                    if(vectorImagen[i
    -257] < vectorImagen[i]) hashPixel
     += 1;
                    hashPixel *= 2;
                    if(vectorImagen[i
    -256] < vectorImagen[i]) hashPixel
     += 1;
                    hashPixel *= 2;
                    if(vectorImagen[i
    -255] < vectorImagen[i]) hashPixel
     += 1;
                    hashPixel *= 2;
                    if(vectorImagen[i
    + 1] < vectorImagen[i]) hashPixel
    += 1;
                    hashPixel *= 8; //
    shift left 3
                    if(vectorImagen[i
    - 1] < vectorImagen[i]) hashPixel
    += 1;
                }
            }else{//esta en la carnita
     de la imagen
                if(i%256 == 0){//si
    esta en la fila de la izuierda
                    if(vectorImagen[i
    -256] < vectorImagen[i]) hashPixel
     += 1; //pixel arriba
                    hashPixel *= 2;
                    if(vectorImagen[i
    -255] < vectorImagen[i]) hashPixel
     += 1; // esquina superior derecha
                    hashPixel *= 2;
                    if(vectorImagen[i
    + 1] < vectorImagen[i]) hashPixel
    += 1; //pixel derecha
                    hashPixel *= 2;
                    if(vectorImagen[i
    +257] < vectorImagen[i]) hashPixel
     += 1; // esquina inf derecha
                    hashPixel *= 2;
                    if(vectorImagen[i
    +256] < vectorImagen[i]) hashPixel
     += 1; // esquina superior derecha
                    hashPixel *= 4;//
    shift left 2
                }else if(i%256 == 255)
    {//Esta en la fila de la derecha
                    if(vectorImagen[i
    -257] < vectorImagen[i]) hashPixel
     += 1; //pixel superior izq
                    hashPixel *= 2;
                    if(vectorImagen[i
    -256] < vectorImagen[i]) hashPixel
     += 1; //pixel arriba
                    hashPixel *= 16;
                    if(vectorImagen[i
    +256] < vectorImagen[i]) hashPixel
     += 1; // pixel abajo
                    hashPixel *= 2;
                    if(vectorImagen[i
    +255] < vectorImagen[i]) hashPixel
     += 1; // pixel inf izq
                    hashPixel *= 2;
                    if(vectorImagen[i
    - 1] < vectorImagen[i]) hashPixel
    += 1; // pixel izquierda
                }
            }
```

```
88                histograma[hashPixel] +=
    1;
89            }
90            long hash = productoPunto(
    histograma, "lbp");
91            Bucket b = sing.getBucketLBP(
    hash);
92            if(b==null){
93                b = new Bucket(Long.
    toString(hash),hash);
94                sing.insertarBucketLBP(b);
95            }
96            b.agregarImagen(bitmap);
97            actualizarListView();
98        }
99
100
101
```

3.2 Analisis O grande

La O grande del algoritmo LBP es de $O(256 * m + 65536 * 2)$, puesto que para crear el histograma es necesario recorrer cada pixel de la imagen y calcular el número hash que se forma con los pixeles vecinos. En este caso m es el numero de hiperplanos con los que se compara el algoritmo al efectuar el LSH. De esta forma se concluye que: a pesar de que la eficiencia formal es linear $O(256 * m)$ se debe dejar claro que en cualquier caso siempre se van a efectuar al menos 65536 pasos en el algoritmo.

4 LSH

4.1 Descripción

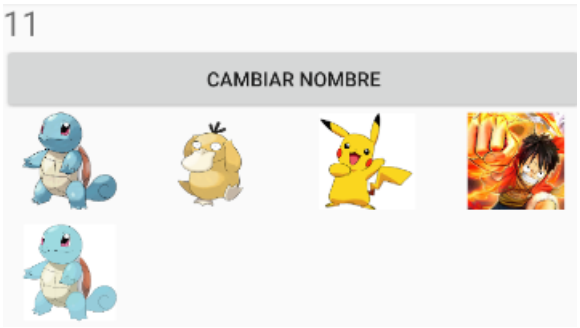
La aplicacion utiliza un algoritmo de Local Sensitive Hashing que se encarga de que al recibir una imagen, de formato bitmap, este devuelva un Long con el numero de hash que a este le corresponde. Para el uso de este algoritmo es necesario el uso de K planos, lo que significa que habran 2^K posibilidades de hashes. El LSH se implementa utilizando el producto punto de los hiperplanos del algoritmo con el del vector que se está categorizando. El algoritmo es el mismo que se conoce generalmente, y muy sencillo de programar ya que solamente es hacer la sumatoria de la multiplicacion entre todos los valores de cada indice de los vectores.

4.2 Análisis O Grande

La O grande del algoritmo LSH implementado es $O(n * m)$, donde m es la cantidad de planos con los que se va a clasificar el vector de entrada y n la cantidad de dimensiones del vector que se esta comparando. Por lo tanto, cuando se analiza el histograma de texturas del algoritmo LBP se maneja $n = 256$ normalmente, mientras que en el de Pixeles $n = 65536$ para imagenes de 256x256

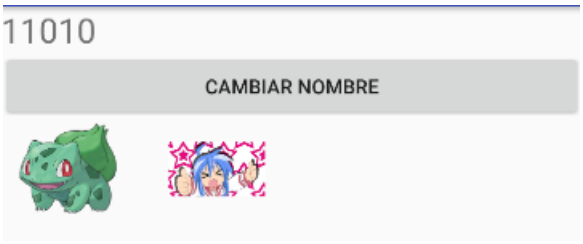
```
1
2   public long productoPunto(int[]
     array1, String tipo){
3         int mull = 1;
4         long result=0;
5         int contador = 256;
6         if(tipo.equals("pixels"))
    contador = 65536;
```

```
7           for(int i = 0; i< cantBuckets; i++){
8               long dot =0;
9               int rand[] = randoms.get(i);
10              for(int e=0; e<contador; e++){
11                  dot += array1[e] * rand[e];
12              }
13              if(dot%2 == 0)
14                  result += 1*mull;
15              mull*=10;
16          }
17          return result;
18      }
19
20
```



En este hash hay imagenes donde se nota que solo dos de ellas se parecen, pero esto se dice que es por la poca cantidad de diferentes hash que se usan. Se espera que aumentando el numero de planos esto mejore.

Hash de 5 digitos:



Cuando se utiliza un hash de 5 digitos, lo esperado es que las dos imagenes de squirtle queden en el mismo bucket, y que las otras que no se parecen queden en buckets separados. Pero este no es el caso, misteriosamente los squirtles quedan separados, las unicas imagenes que quedan en un mismo bucket son las que se encuentran anteriormente. Estas imagenes no tienen mucho parecido para nosotros, pero el algoritmo asi las clasifica.

5 Experimentos

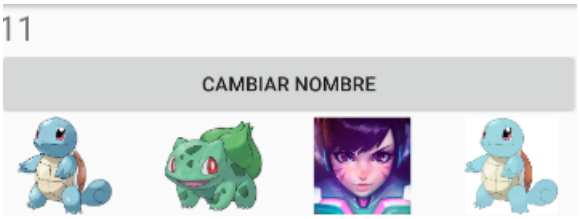
Uno de los experimentos fue probar con distintos largos de hash para descubrir si la calidad del programa cambia con respecto a la cantidad de digitos que tenga el mismo. Con ambos algoritmos se hicieron las distintas pruebas cambiando el largo del hash. Para pixels se usó 2 y 5 de largo y para LBP se usó 2 y 10. Fotos tomadas con el emulador de Android.

5.1 Pixels:

Hash de 2 digitos:

5.2 LBP:

Hash de 2 digitos:



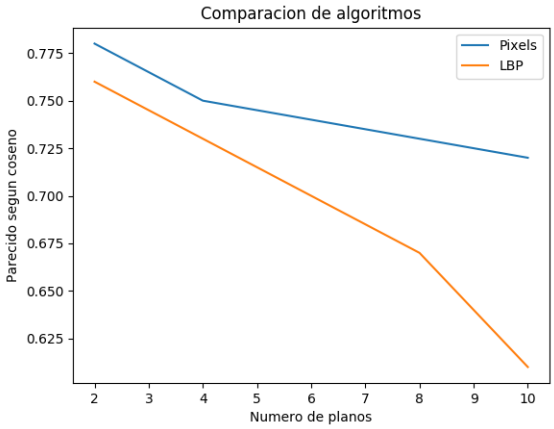
Utilizando el lbp con 2 de largo en los hash, igual que en pixels los buckets tienen muchas imagenes y la mayoría de ellas no tienen algun parecido. Pero de nuevo esto es por la poca cantidad de posibilidades de obtener un hash diferente

Hash de 10 digitos:

En este caso si se obtiene el resultado esperado, todas las imagenes se encuentran en buckets separados menos las mostradas anteriormente. Estas se esperaban que quedaran en el mismo bucket desde el inicio porque es la misma imagen solo con algunos variantes en su coloración.

En ambos algoritmos si se aumentaba el largo del hash todas las imagenes quedaban en buckets separados por lo que decidimos dejarlas en ese número.

Luego se genera el siguiente grafico para demostrar formalmente el parecido de las imagenes de los buckets, el numero del parecido es un promedio del parecido de las imagenes dentro del bucket.



6 Conclusión

En conclusion, podemos observar que este tipo de funciones para una aplicacion no son tan complicadas como lo parecen, puesto que los algortimos existentes para esto tienen una logica similar y sencilla. Y con el uso basico de las mismas podemos crear algo tan funcional como lo es esta aplicacion para repartir imagenes en sus carpetas correspondientes con imagenes parecidas.