Algoritmos para el análisis de formas y reconocimiento de patrones bitonales

Una implementación en sintaxis de Processing (Java)

Autor: Emiliano Causa

e_causa@yahoo.com.ar , www.biopus.com.ar , www.emiliano-causa.com.ar

Palabras claves

Análisis de formas, reconocimiento de patrones bitonales, captura de movimiento, visión artificial

Resumen

En este trabajo se plantearán los algoritmos de análisis de formas bitonales, con el fin de poder discriminar objetos que se mueven en forma independiente, en los procesos de captura de movimiento por substracción de video. Este algoritmo intenta solucionar los problemas más comunes que surgen en este tipo de captura de movimiento cuando se enfrenta a situaciones en las que existen más de un objeto moviéndose en forma independiente. Con el fin de estudiar dicho algoritmo, se profundizará en cada una de las etapas necesarias para llevar a cabo el algoritmo, particularmente los algoritmos necesarios para analizar las sub-regiones y regiones de la imagen, así como su grado de vecindad, inclusiones y niveles.

1. Introducción

Los algoritmos de "análisis de forma" que se tratarán en este texto fueron desarrollados por el autor con el fin de poder discriminar ciertas características de las formas en las imágenes bitonales.

1.1. Captura de movimiento por substracción de video y el problema de los objetos independientes

Los procesos de captura de movimiento por substracción de video, devuelven imágenes bitonales que representan, en blanco, al sujeto del movimiento (lo que se está moviendo), y en negro, a lo que se encuentra quieto. En las situaciones en donde existe un único objeto en movimiento, el cálculo de posición que el algoritmo de captura de movimiento (tratado en mi anterior trabajo "Algoritmos de captura óptica de movimiento por substracción de video"), resulta preciso, pero cuando existe más de un objeto en movimiento (como ocurre con dos personas caminando frente a la cámara), dichos cálculos dejan de ser correctos. Esto sucede por que este tipo de algoritmo toma todo lo que se mueve como una unidad, es decir, no puede discriminar entre diferentes objetos que se mueven en

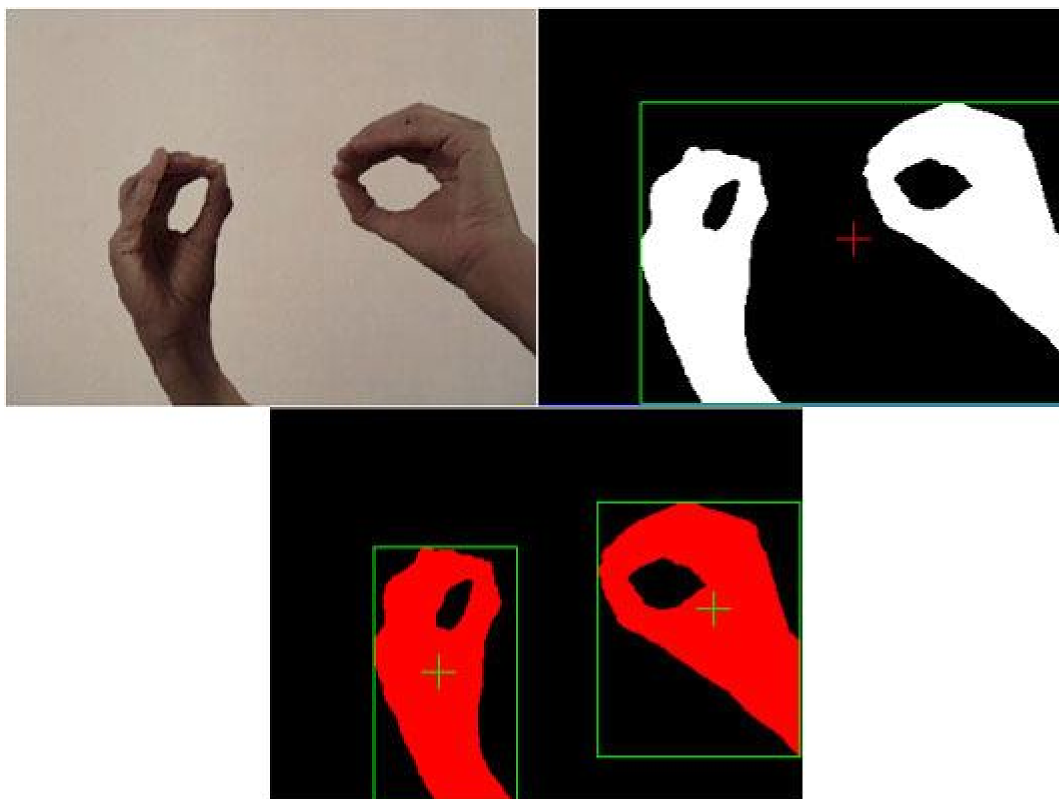
forma independientes unos de otros. La principal dificultad para esta discriminación consiste en que el algoritmo sólo separa píxeles pertenecientes al movimiento de los que no pertenecen, sin embargo, no existe una diferenciación con respecto a cuál objeto genera ese movimiento.

Figura 1: captura del movimiento mediante substracción de video. La imagen de la izquierda es el fondo, la de la derecha es un fotograma de la escena actual y debajo el resultado de la captura de movimiento.

Este tipo de captura de movimiento toma como posición del movimiento, al promedio de la posición de los píxeles del área de movimiento (los píxeles blancos). En la **figura 1** se puede ver que la posición que devuelve el algoritmo de captura de movimiento (indicado en la imagen por una cruz roja) es correcto y representa (a groso modo) la posición central de la mano.

En la **figura 2** se pueden ver dos manos que se mueven en forma independiente, la figura del extremo superior derecho (las manos blancas) muestra que el cálculo de posición (cruz roja) no es el adecuado, dado que no es la posición de ninguna de las dos manos, sino el centro de las dos manos tomados como una mancha indiscriminada, en cambio, en la parte inferior se puede ver lo que debería ser un correcto cálculo de posiciones (cruces verdes sobre manos rojas). Este tipo de análisis, sólo se puede realizar mediante una correcta discriminación de los píxeles que pertenecen a una mano con respecto a los de la otra.

Figura 2: captura del movimiento de dos objetos separados. Arriba izquierda: imagen original. Arriba derecha: captura de movimiento sin análisis de forma. Abajo: captura de movimiento con análisis de forma



1.2. Captura de regiones cerradas

Más allá de resolver el problema de los “objetos independientes”, el algoritmo de “análisis de formas” permite también analizar los distintos niveles de inclusión. Por ejemplo, puede suceder que dentro de una forma blanca se encuentre encerrada una forma negra, este caso se da en la imagen que muestra la **figura 3**, en donde las formas blancas de la manos encierran figuras negras (resultantes de cerrar los dedos en forma de pinza). Este tipo de análisis le ha permitido al grupo **mine-control** (www.mine-control.com) realizar obras en las cuales las personas interactúan usando su sombra y utilizan sus manos en forma de pinza como punteros (como se puede ver en su trabajo “Fractal zoom”, **figura 4**).

Figura 3: análisis de formas de diferentes niveles de inclusión. Derecha: formas de nivel 1. Izquierda: formas de nivel 2

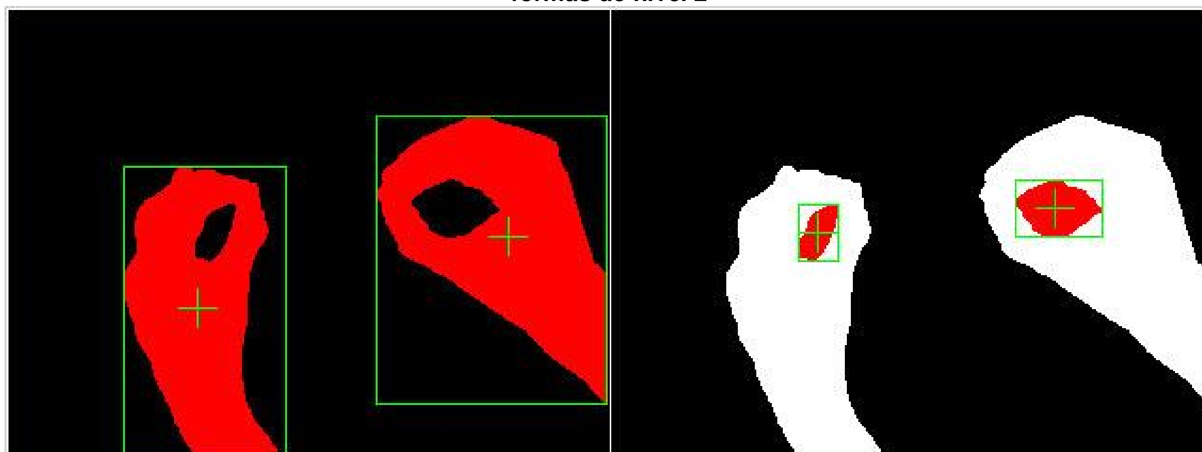
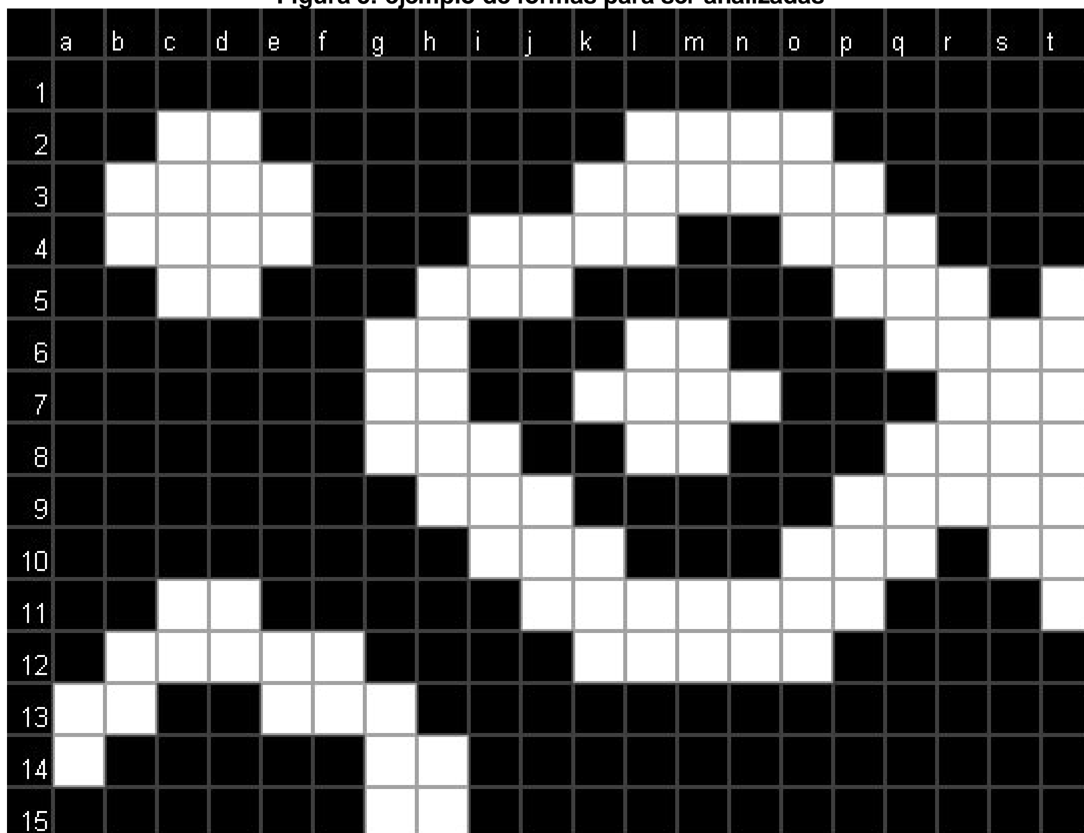


Figura 4: Fractal Zoom del grupo “mine-control” (imagen obtenida de www.mine-control.com)

2. Algoritmo de Análisis de formas

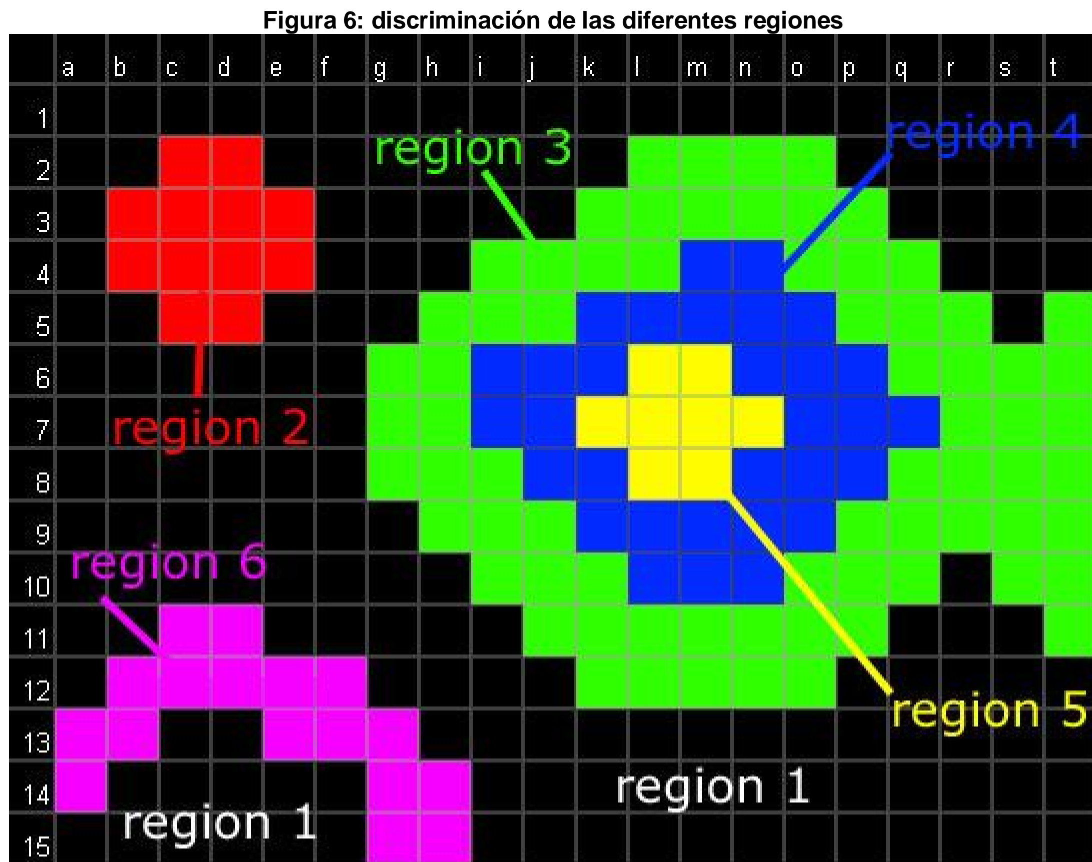
Figura 5: ejemplo de formas para ser analizadas



El Algoritmo de Análisis de Formas (AAF a partir de ahora), recibe una imagen bitonal (blanco y negro) y a partir de esta analiza las regiones buscando los píxeles conexos e inconexos. Por ejemplo,

en la **figura 5** podemos ver un ejemplos de imagen bitonal que hemos ampliado y le agregamos números y letras para identificar las filas y columnas (respectivamente) de cada píxel. Esto nos permitirá seguir el proceso del algoritmo píxel por píxel.

Figura 6: discriminación de las diferentes regiones



En la **figura 6**, mostramos como fueron identificadas cada una de las regiones en función de las conexiones entre los píxeles. Por ejemplo: la **Región 1** está conformada por todos los píxeles negros que son vecinos del píxel a1, así como todos los píxeles negros que son vecinos de alguno de estos vecinos. Por ejemplo, el píxel b1 es vecino de a1 y como es negro lo vamos a considerar “conexo” con a1. El píxel c1 es vecino de b1 (el píxel conexo con a1) y como también comparte su color (es negro) entonces lo consideramos conexo con b1 y por ende es conexo con a1. La regla sería: todo píxel Y vecino de un píxel X si comparte su color se dice que “Y es conexo con X”, por ley transitiva: todo píxel Z vecino de un píxel Y conexo con X y que comparte el color con ambos se dice que “Z es conexo con Y” y “Z es conexo con X”. Por ende:

la **Región 1** está conformada por todos los píxeles conexos con a1,

la **Región 2**, por todos los píxeles conexos con c2,

la **Región 3**, por todos los píxeles conexos con l2,

la **Región 4**, por todos los píxeles conexos con m4,

la **Región 5**, por todos los píxeles conexos con l6,

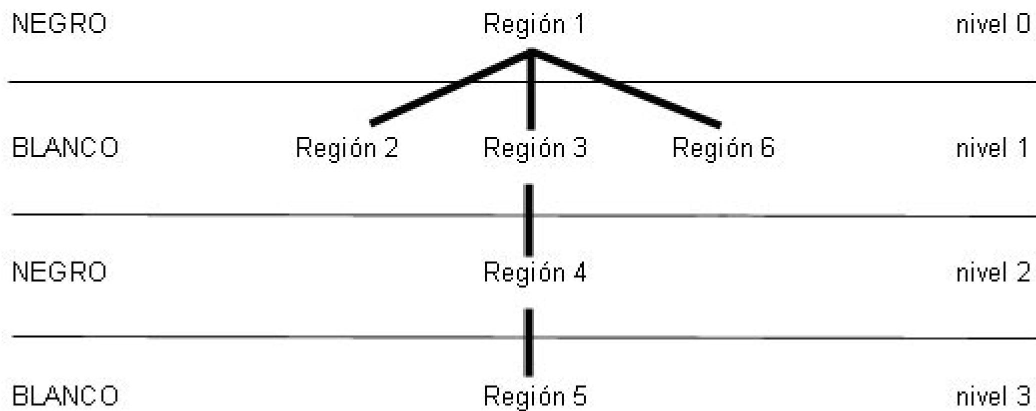
la **Región 6**, por todos los píxeles conexos con c11,

la **Región 7**, por todos los píxeles conexos con c13,

Si bien una región podría ser definida por los píxeles conexos con cualquiera de sus píxeles, es decir que cualquiera de sus píxeles podría ser elegido como píxel de referencia, decidí elegir (en el párrafo anterior) los píxeles que aparecen primeros en el recorrido de la imagen, que se realiza en el orden de la lectura (de izquierda a derecha y de arriba hacia abajo).

Otro dato que revela el análisis son las relaciones de inclusiones que se dan entre las regiones. Como muestra la **figura 7**, se puede hacer un árbol de inclusiones, en donde el primer nivel (llamado nivel 0) es la región negra que toca los bordes, el nivel 1 son las figuras blancas incluidas en las regiones de nivel 0 (como la **región 2**) o que tocan algún borde (como las **regiones 3 ó 6**), el nivel 2 son las regiones incluidas en las regiones del nivel 1, y así sucesivamente:

Figura 7: árbol de inclusiones de las regiones de la figura 5



El AAF debe seguir una serie de pasos para analizar los datos recién descriptos. En el algoritmo siguiente se pueden ver dichos pasos:

Algoritmo 1: Algoritmo Principal de Análisis de Formas

```
void analizarRegiones(){
    buscarSubRegiones();
    buscarRegiones();
    revisarLimitesDeRegiones();
    analizarVecindad();
    analizarInclusiones();
    analizarNiveles();
}
```

2.1. Búsqueda de sub-regiones

El primer paso para el AAF es la “búsqueda de sub-regiones”. Dicho paso consiste en recorrer la imagen píxel por píxel siguiendo el orden de la lectura, e ir revisando las relaciones de vecindad de cada uno. La **figura 9** muestra la forma en que se establecen las sub-regiones de cada uno de los píxeles.

Píxel a1: como no tiene vecinos se establece la primer sub-región (la **1**). La forma en que se revisa los vecinos es siguiendo el orden que muestra la **figura 8**: primero el píxel ubicado al Oeste, luego al Nord Oeste, luego al Norte y luego al Nord Este. Esto es por que esos son los píxeles que tienen asignados sub-región, dado que son los píxeles ya recorridos, según el orden de recorrido de la imagen –izquierda a derecha de arriba hacia abajo-.

Píxel b1: su único vecino con sub-región asignada es a1 y como ambos píxeles son negros entonces toma la misma sub-región.

Píxel c1: igual que el caso anterior su vecino es el píxel b1 y al ser también negro toma la sub-región 1.

Desde los píxeles d1 hasta t1: se repite el caso anterior.

Píxel a2: el primer vecino con sub-región asignada es a1 y por ende toma la sub-región 1 (a la que pertenece a1).

Píxel b2: toma del a2.

Píxel c2: es el primer píxel blanco y por ende ninguno de sus vecinos sirve para tomar de referencia y usar su sub-región, por ende se genera una nueva sub-región (la **2**).

Píxel d2: lo toma de c2, ya que es el único vecino con el mismo color (blanco).

Píxel e1: toma la sub-región del primer vecino de su mismo color (negro), que es el d1.

Este proceso se repite y se logra asignar las sub-regiones de la forma expuesta en la **figura 9**. Si se observa con detenimiento se puede ver que existen sub-regiones que pertenecen a una misma región (en la figura 6), este es el caso de:

las sub-regiones **1** y **10** (de la figura 9) que pertenecen la región **1** (de la figura 6)

las sub-regiones **3, 4** y **7** (de la figura 9) que pertenecen la región **3** (de la figura 6)

las sub-regiones **5, 6** y **8** (de la figura 9) que pertenecen la región **4** (de la figura 6)

Esta situación surge como resultado de la forma en que se revisa la vecindad (como muestra la **figura 8**) y como resultado del recorrido de la imagen. Para no llegar a esta situación habría que hacer otro tipo de recorrido más exhaustivo de la imagen pero que también conllevaría una baja de la performance del algoritmo. La forma en que se resuelve este problema es haciendo que el algoritmo asigne un alias a las sub-regiones que pertenecen a la misma región.

Así, la sub-región **10** es un alias de la sub-región **1,**

las sub-regiones **4** y **7** son alias de la sub-región **3,**

y las sub-regiones **6** y **8** son alias de la sub-región **5**.

Esto facilita las posterior identificación de regiones, dado que una región esta conformada por su sub-región y todos sus “alias”.

Figura 8: revisión e vecindad

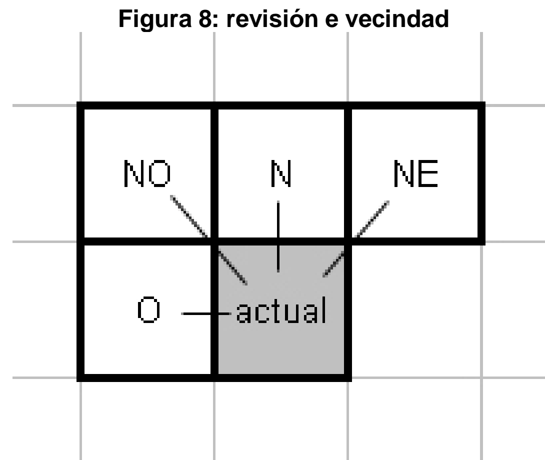


Figura 9: identificación de sub-regiones

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	2	2	1	1	1	1	1	1	1	3	3	3	3	1	1	1	1	1
3	1	2	2	2	2	1	1	1	1	1	3	3	3	3	3	3	1	1	1	1
4	1	2	2	2	2	1	1	1	4	4	4	4	5	5	3	3	3	1	1	1
5	1	1	2	2	1	1	1	4	4	4	6	6	6	6	6	3	3	3	1	7
6	1	1	1	1	1	1	4	4	8	8	8	9	9	6	6	6	3	3	3	3
7	1	1	1	1	1	1	4	4	8	8	9	9	9	9	6	6	6	3	3	3
8	1	1	1	1	1	1	4	4	4	8	8	9	9	6	6	6	3	3	3	3
9	1	1	1	1	1	1	1	4	4	4	8	8	8	8	8	3	3	3	3	3
10	1	1	1	1	1	1	1	1	4	4	4	8	8	8	3	3	3	10	3	3
11	1	1	11	11	1	1	1	1	1	4	4	4	4	4	4	4	10	10	10	3
12	1	11	11	11	11	11	1	1	1	1	4	4	4	4	4	10	10	10	10	10
13	11	11	12	12	11	11	11	1	1	1	1	1	1	1	1	1	1	1	1	1
14	11	12	12	12	12	12	11	11	1	1	1	1	1	1	1	1	1	1	1	1
15	12	12	12	12	12	12	11	11	1	1	1	1	1	1	1	1	1	1	1	1

El procedimiento que acabamos de describir lo realiza el siguiente algoritmo llamado Buscar Sub-Regiones:

Algoritmo 2: Buscar Sub-Regiones

```
void buscarSubRegiones(){
   cantSub=0;
   int primer;
   int ulti1=0,ulti2=0;
   for(int j=0;j<imagen.alto;j++){ //lin 5: recorre la imagen a lo alto
      for(int i=0;i<imagen.ancho;i++){ //lin 6: recorre la imagen a lo ancho
         primer=primer_vecino(i,j); // lin 7: busca el primer vecino del mismo color
         if(primer==0){ // lin 8: si no existe dicho vecino
            //agrega una nueva subregion
            cantSub++;
            subReg[i][j] = cantSub; // lin 11: asocial el píxel a la nueva region
         }else{
            //toma la region del primer vecino
            int region_primero = subRegion_de_vecino(i,j,primer); //
            subReg[i][j] = region_primero; // lin 15: asocia al píxel a la región del primer vecino
            for(int k=primer+1 ; k<=4 ; k++){ // lin 16: recorre el resto de los vecinos
               if( vecino_es_igual(i,j,k) ){ // lin 17: si el nuevo vecino recorrido es del mismo color
                    int vecino=subRegion_de_vecino(i,j,k); // lin 18: averigua la region de dicho vecino
                  //corrige las otras regiones vecinas
                    if(region_primero !=  vecino){ // lin 20: si la region no coincide con la que tomo
                                                   //del primer vecino
                                                   //decide generar un alias
                       if(!(vecino==ulti1 && region_primero==ulti2)){ // lin 23: revisa que la asociacion
                                                                      //no se haya hecho recién
                          if( este_alias_no_esta_asociado(vecino,region_primero) ){ // lin 25: revisa que la
                                                        //asociación no se haya hecho antes
                              if(subAlias[vecino]==0){ // lin 27
                                 //si todavia no está asociada le asigna un alias
                                 subAlias[vecino]=region_primero;
                                 ulti1=vecino;
                                 ulti2=region_primero;
                              }else if(subAlias[vecino]!=region_primero){ // lin 32
                                 //como ya está asociado busca el final de la cadena
                                 //y asocia el último eslabón
                                 int alias_profundo = devolver_alias(vecino);
                                 subAlias[alias_profundo]=region_primero;
                                  ulti1=vecino;
                                  ulti2=region_primero;
                              }
                          }
                       }
                    }
                 }
              }
           }
        }
     }
   }
```

Líneas 5 y 6 (las cuales tienen comentarios etiquetados “//lin 5:” y “//lin 6:” respectivamente): los ciclos recorren la imagen de izquierda a derecha y de arriba hacia abajo.

Línea 7: se busca al primer vecinos del píxel, es decir al primer píxel dentro de la vecindad desripta en la **figura 8**, y que coincida en color con el píxel actual. La función **int primer_vecino(x , y)** retorna un número entre 0 y 4, que representa:

0	no existe vecino
1	vecino Oeste
2	vecino Nord Oeste
3	vecino Norte
4	vecino Nord Este

Líneas 10 y 11: cuando en la línea 8 evalúa que no existe un vecino, genera una nueva sub-región (incrementando el contador de sub-regiones, **cantSub**) y luego asocia el píxel a dicha sub-región usando la matriz creada para ese fin (**subReg [x] [y]**).

Línea 15: en caso de existir un vecino, se carga en la matriz **subReg [x] [y]**, el número de sub-región del primer vecino.

Línea 16: revisa el resto de los vecinos con el fin de buscar otro que posea el mismo color pero que por razones de orden de recorrido tenga asignado otra sub-región, caso en el cual hay que asociar ambas sub-regiones con un alias.

Línea 17: la función **bolean vecino_es_igual(x , y , vecino)** devuelve **true** cuando dicho vecino es del mismo color.

Línea 18: averigua la sub-región a la que pertenece dicho vecino.

Línea 20: en esta línea se evalúa si dos vecinos iguales tienen asignados diferentes sub-regiones, caso en el que habría que establecer un alias. Este caso se da en el píxel j4 (del ejemplo), en donde los vecinos oeste (píxel i4) y el nord este (píxel k3) poseen sub-regiones diferentes asignadas, la **3** y la **4**, lo que indica que las sub-región 4 es un alias de la sub-región 3.

Línea 23 y 25: como es necesario generar el alias, se revisa que este no haya sido hecho antes.

Línea 27 a 31: como la región no posee alias, se le asigna el alias en cuestión.

Línea 32 a 38: si la región ya posee un alias, busca, en la cadena de alias, el más profundo y lo pone como alias del actual.

2.2. Búsqueda de regiones

Una vez que el algoritmo encargado de analizar las sub-regiones finaliza, empieza una nueva etapa: la de juntar las sub-regiones y sus alias en una única región. Este algoritmo recorre la imagen píxel por píxel y les asigna la región que le corresponde, uniendo todos los alias.

Algoritmo 3: Buscar Regiones

```
void buscarRegiones(){
    int ulti1=0,ulti2=0;
    int cual,deQuien;
    cantRegi=0;
    for(int j=0;j<imagen.alto;j++){ // lin 5: recorre la imagen
      for(int i=0;i<imagen.ancho;i++){ // lin 6: recorre la imagen
        cual = getSubRegion(i,j); // lin 7: encuentra la sub-región o alias asociado al píxel
        if(cual != ulti1){ // lin 8: si difiere del último
          deQuien = aliasPerteneceARegion(cual); // lin 9: busca la región asociada a la sub-región
          if(deQuien>0){ // lin 10: si la región existe
            eReg[ deQuien ].agregarPixel(i,j); // lin 11: agrega el píxel a la región
            pixReg[i][j] = deQuien; // lin 12: vincula el píxel a la región
          }
          else{
            cantRegi++; // lin 15: cuenta una nueva región
            eReg[cantRegi] = new AM_Region( cantRegi ); // lin 16: crea una nueva región
            eReg[cantRegi].esBlanco=(original.getPixel(i,j)==1); // lin 17: carga el color
                                                                // de la región
            pixReg[i][j] = cantRegi; // lin 19: vincula el píxel a la región
            eReg[cantRegi].alias = cual; // lin 20: vincula la región al alias de la sub-región
            ulti1=cual; // lin 21: memoriza la última sub-región
            ulti2=cantRegi; // lin 22: // memoriza la última región
            eReg[cantRegi].agregarPixel(i,j); // lin 23: agrega el píxel a la región
          }
        }
        else{ //lin 26: si no difiere del último entonces
          eReg[ulti2].agregarPixel(i,j); // lin 27: usa los últimos datos
          pixReg[i][j] = ulti2; // lin 28: para vincular el píxel a la region
        }
      }
    }
  }
```

Líneas 5 y 6: los ciclos recorren la imagen de izquierda a derecha y de arriba hacia abajo.

Línea 7: encuentra la sub-región o alias asociado al píxel

Línea 8 y 9: si difiere de la última región y sub-región usada, busca la región a asociada a la sub-región

Línea 10 a 12: si existe la región, agrega el píxel a la región y vincula el píxel a la región

Línea 15 a 23: si la región no existe, entonces crea una nueva región y carga sus parámetros, vincula el píxel a la región, vincula la región al alias de la sub-región y agrega el píxel a la región.

Línea 26 a 28: si no difiere de la última región y sub-región usada, usa los últimos datos para vincular el píxel a la región.

Una función importante en este algoritmo es la llamad a **void agregarPixel(int x , int y)**, la cual es un comportamiento de la clase **AM_Region.** Esta función se encarga de actualizar los atributos de la región: superficie, bordes, centro, dimensiones:

Algoritmo 4: Agregar Píxel

```
void agregarPixel(int x,int y){
  if(sup==0){
    top=botton=y;
    left=right=x;
    ancho=1;
    alto=1;
  }
  else{
    if(x<left){
      left=x;
      ancho=right-left+1;
    }
    if(x>right){
      right=x;
      ancho=right-left+1;
    }
    if(y<top){
      top=y;
      alto=botton-top+1;
    }
    if(y>botton){
      botton=y;
      alto=botton-top+1;
    }
  }
  sup++;
  centroX = centroX * ((sup-1)/sup) + float(x) / sup;
  centroY = centroY * ((sup-1)/sup) + float(y) / sup;
}
```

2.3. Revisión de límites

Cuando el algoritmo “buscarRegiones” finaliza y cada uno de los píxeles tiene su región de pertenencia, y cada región tiene definidos sus atributos. El siguiente paso se trata de revisar los límites de las regiones con el fin de establecer cuáles de estas tocan el borde de la imagen y cuales no. Esto es necesario para establecer en el siguiente paso el árbol de inclusiones, dado que el primer nivel (el nivel 0) está conformado por las regiones negras que tocan el borde (como la **región 1** del ejemplo), el nivel 1 son las regiones blancas que tocan el borde (como las **regiones 3 y 6** del ejemplo) y también las regiones incluidas en las del nivel 0. El resto de los niveles se definen por inclusiones, las cuales se deducen de las vecindades.

La función que logra revisar los límites es sencilla:

```
void revisarLimitesDeRegiones(){
    for(int i=1;i<=cantRegi;i++){
      eReg[i].tocaBorde = (eReg[i].top<=margenBorde || eReg[i].left<=margenBorde
         || eReg[i].right>=original.ancho-1-margenBorde
         || eReg[i].botton>=original.alto-1-margenBorde);
    }
}
```

2.4. Análisis de vecindad

El siguiente algoritmo se encarga de recorrer la imagen píxel por píxel buscando píxeles vecinos que pertenezcan a diferentes regiones, cuando encuentra este caso, registra a dichas regiones como vecinas:

Algoritmo 5: Analizar Vecindad

```
void analizarVecindad(){
    int ulti1=0,ulti2=0;
    int regVec;
    int esteReg=0;
    for(int j=0;j<imagen.alto;j++){ // lin 5: recorre la imagen
      for(int i=0;i<imagen.ancho;i++){ // lin 6: recorre la imagen
        esteReg = pixReg[i][j]; // lin 7: obtiene la region del píxel
        for(int k=1;k<=4;k++){ // lin 8: recorre los vecinos
          regVec = region_de_vecino(i,j,k); // lin 9: obtiene la región de este vecino
          if(ulti1!=esteReg && ulti2!=regVec){ // lin 10: si la relación entre vecino y el píxel
                                              // cambia del último
            if( esteReg != regVec && regVec!=0){ // lin 12: revisa que los vecinos sean de
                                                // distintas regiones y que el vecino no
                                                // sea cero (los puntos del borde)
              ulti1=esteReg;
              ulti2=regVec;
              eReg[regVec].agregarVecino(esteReg); // lin 17: vincula a los vecinos
              eReg[esteReg].agregarVecino(regVec); // lin 18: vincula a los vecinos
            }
          }
        }
      }
    }
  }
```

Líneas 5 y 6: los ciclos recorren la imagen de izquierda a derecha y de arriba hacia abajo.

Línea 7 a 9: obtiene la región del píxel actual y de sus vecinos

Línea 10 a 12: si no se ha revisado este caso aún y las regiones no coinciden

Línea 17 y 18: relaciona a las regiones vecinas

2.5. Análisis de inclusiones

En el algoritmo que sigue, se analizan las inclusiones. Para ello, se recorren las regiones buscando aquellas que tocan el borde, a las cuales se las considera como “no incluidas” en otras, este sería el caso de las regiones 1, 3 y 6 del ejemplo. Para las regiones que no tocan el borde, se revisan sus vecinos verificando quién posee bordes más expandidos (cuál es más grande), por ejemplo la región 5 del ejemplo, estaría incluida en la región 6 por cumplir dos requisitos:

1) Las regiones 5 y 6 son vecinas
2) La región 5 es más pequeña que la 6

Algoritmo 6: Analizar las inclusiones

```
void analizarInclusiones(){
  int vecino;
  for(int i=1;i<=cantRegi;i++){ // lin 3: recorre las regiones
    if(eReg[i].tocaBorde){ // lin 4: si toca el borde se la considera no incluida
      eReg[i].incluidaEn = 0;
    }
    else{ // lin 7: si no toca el borde
      for(int j=1;j<=eReg[i].cantVecinos;j++){ // lin 8: recorre las regiones vecinas
        vecino = eReg[i].vecinos[j]; // lin 9: recorre los vecinos
        if( estaIncluido(i,vecino) ){ // lin 10: revisa la inclusión en función de los bordes
          eReg[i].incluidaEn=vecino; // lin 11: si esta incluida lo registra
          break;
        }
      }
    }
  }
}
```

La función **boolean estaIncluido(int regionDentro , int regionFuera)** se encarga de revisar los tamaños de los vecinos:

```
boolean estaIncluido(int regionDentro,int regionFuera){
  int a=regionDentro;
  int b=regionFuera;
  return (eReg[a].top >= eReg[b].top && eReg[a].left >= eReg[b].left
      && eReg[a].right<=eReg[b].right && eReg[a].botton <= eReg[b].botton);
}
```

2.6. Análisis de niveles

Por último, el algoritmo 7, para el análisis de niveles, revisa las regiones siguiendo el siguiente criterio. Primero revisa aquellas regiones que tocan el borde o que no están incluidas en ninguna otra, a estas regiones les asigna el nivel 0 y el nivel 1 dependiendo de si son negras o blancas respectivamente, esto se realiza entre las líneas 5 hasta la 20 (inclusive). La segunda parte del algoritmo toma una a una las regiones (línea 23), revisando si ya tiene nivel asignado (línea 26), luego revisa todas la regiones sin nivel asignado (línea 29), buscando aquellas que están incluidas en la primera (línea 30), a las cuales le asigna un nivel más que el de sus padre (la primera región) (línea 32).

Algoritmo 7: Analizar niveles

```
void analizarNiveles(){
    int faltan=cantRegi;
    int nuevoNivel;
    int nivelPadre;
    for(int i=1;i<=cantRegi;i++){ // lin 5: recorre las regiones
        if(eReg[i].tocaBorde || eReg[i].incluidaEn==0){ // lin 6: si toca el borde o no está incluida
                faltan--; // lin 7: descuenta una región
                if(eReg[i].esBlanco){ // lin 8: si la región es blanca
                        eReg[i].nivel=1; // lin 9: la considera nivel 1
                        (cantXNivel[1])++;
                }
                else{
                        eReg[i].nivel=0; // lin 13: si es negra la considera nivel 0
                        (cantXNivel[0])++;
                }
        }
        else{
                eReg[i].nivel=-1; // lin 18: a las que no son nivel 0 ó 1 les asigna -1 para tratar después
        }
    }
    //mientras falte asignar niveles a regiones
    for(int j=0;faltan>0 && j<cantRegi*3;j++){ // lin 22: mientras falte asignar niveles a algunas regiones
        for(int i=1;i<=cantRegi;i++){ // lin 23: recorre las regiones
                //para todas las regiones con niveles
                nivelPadre=eReg[i].nivel;
                if(nivelPadre>=0){ // lin 26: toma una región (i) con nivel ya asignado
                        for(int k=1;k<=cantRegi;k++){
                        //revisa si las que no tienen niveles (k) están incluidas en esta (i)
                                if(eReg[k].nivel==-1){ // lin 29: toma una región (k) sin nivel asignado
                                        if(eReg[k].incluidaEn==i){ // lin 30: si la región (k)
                                                                    // está incluida en la región (i)
                                                nuevoNivel=nivelPadre+1; // lin 32: asigna el nivel
                                                eReg[k].nivel=nuevoNivel;// correspondiente
```

```
                                                    if(nuevoNivel<10){
                                                            (cantXNivel[nuevoNivel])++;
                                                    }
                                                    faltan--; // lin 37: descuenta la región
                                            }
                                    }
                            }
                    }
            }
    }
}
```

2.7. Consulta de regiones en el programa principal

Una vez realizados los algoritmos recién descriptos, sólo queda realizar unas pocas funciones que permitan recorrer las regiones resultantes del análisis. Por ejemplo, a continuación podemos ver un ejemplo de un objeto de tipo "AnalisisDeForma" que posee todos nuestros algoritmos dentro. Después de inicializarse con los valores de "superficieMinima" y "nivel", ejecutamos el comando "analizar(imagen)", esta función ejecuta todos los pasos que acabamos de describir:

```
AnalisisDeForma anaForma = new AnalisisDeForma();
anaForma.superficieMinima = 10;
anaForma.nivel = 1;
anaForma.analizar( imagen );
for( int i=0 ; i<anaForms.cantidadDeRegiones() ; i++ ){
        AF_Region estaRegion = anaForma.region( i );
        println( "Region " + i + " ubicada en x=" + estaRegion.x + " y=" + estaRegion.y );
}
```

A lo último, aparece un **ciclo for** que es el encargado de recorrer todas las imágenes de un nivel determinado y retornar dichas regiones a través de un objeto de tipo "AF_Region", que es el que posee información acerca del área de la región, así como su posiciones y límites.

Algoritmo 8: cantidad de regiones de un nivel determinado

```
int cantidadDeRegiones( ){
  return cantidadDeRegiones( nivel );
}
//--------------------------
int cantidadDeRegiones( int esteNivel ){
  int valor = 0;
  for(int i=1;i<=cantRegi;i++){
    if(eReg[i].nivel == esteNivel){
      if(eReg[i].sup>superficieMinima){
        valor++;
```

```
        }
      }
    }
    return valor;
  }
```

El algoritmo 9 muestra dos versiones (sobrecarga) de la función **region( )**, en donde la primera simplemente llama a la segunda usando un nivel por defecto. La segunda recorre todas las regiones (cargadas en el vector **eReg[ ]**), verificando que pertenezcan al nivel que se busca y que posean una superficie mayor a la establecida como mínima. Por último, también aparece una función llamada **regionPorIndice( )** que se encarga de devolver una región en función de su índice en el vector de regiones.

Algoritmo 9: funciones de consulta a una región determinada

```
AF_Region region( int orden ){
  return region( orden , nivel );
}
//--------------------------
AF_Region region( int orden , int esteNivel){
  int valor = 0;
  int cual = -1;
  for( int i=1 ; i<=cantRegi ; i++ ){
    if( eReg[i].nivel == esteNivel ){
      if( eReg[i].sup>superficieMinima ){
        if( valor == orden ){
          cual = i;
          break;
        }
        valor++;
      }
    }
  }
  if( cual != -1){
    return eReg[cual];
  }
  else{
    return null;
  }
}
```

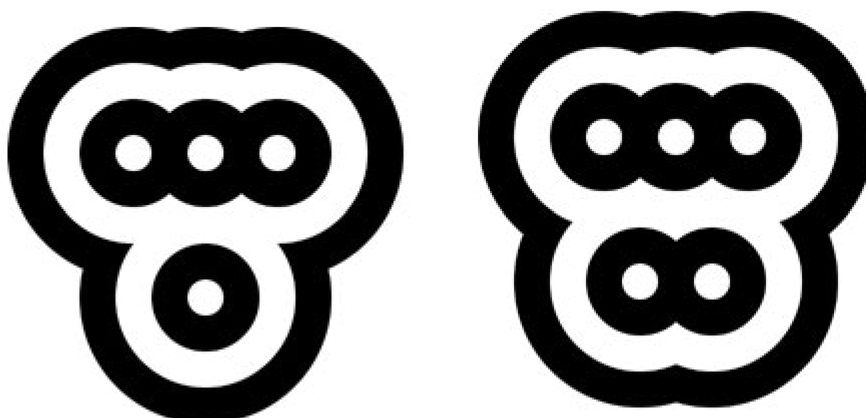
```
//-------------------------
AF_Region regionPorIndice( int indice ){
  return eReg[indice];
}
```

3. Conclusión

Los algoritmos vistos en este trabajo permiten analizar formas bitonales a fin de discriminar sus conexiones e independencias, así como sus relaciones de inclusión. Este tipo de algoritmo es indispensable a la hora de montar pantallas sensibles al tacto (que usan sistemas ópticos de captura de movimiento, mediante tecnología infrarroja), dado que este tipo de dispositivos capta la posición de los dedos a través de diferentes técnicas, que van de la captación de sombra en un tela elástica, hasta la captación de luz mediante "Reflexión Total Interna Frustrada" (*FTIR, frustrated total internal reflection*) (Jefferson Han [1]). Sin embargo, tenemos que señalar que estos algoritmos no son suficientes, dado que sólo son capaces de hacer un análisis de las formas en un instante determinado, es decir en un fotograma por vez, pero no pueden examinar cómo evolucionan estas formas en el tiempo, ese será el tema de mi próximo trabajo en el que concluiremos la relación entre este tipo de algoritmos y la implementación de pantallas sensibles al tacto.

Otra uso que se da a este tipo de algoritmo, es el reconocimiento de patrones construidos con formas bitonales. En la **figura 10** se pueden observar dos patrones construidos con formas bitonales, el que está a la izquierda representa la etiqueta "13", y el otro, la etiqueta "23". Estas etiquetas se leen observando la cantidad de círculos en el nivel más alto de inclusión, que en el primer patrón son 3 círculos en la parte superior y 1 en la parte inferior. El reconocimiento de este tipo de patrones es útil para la detección de objetos físicos en el espacio, dado que se les añade este tipo de patrones y entonces un sistema óptico es capaz de calcular su posición, rotación y distancia.

Figura 10: Patrones bitonales



Esta modalidad de detección requiere de un análisis de formas bitonales, así como el subsiguiente análisis del árbol de inclusiones para detectar las “etiquetas”. Este tipo de algoritmos, será tratado en otro trabajo.

Emiliano Causa

Abril de 2008

4. Referencias bibliográficas

[1] Jefferson Han, *Multi-Touch Interaction Research* (cs.nyu.edu/~jhan/ftirtouch/)

[2] reactable.iua.upf.edu

[3] www.v3ga.net/processing/BlobDetection

[4] opencvlibrary.sourceforge.net

[5] www.processing.org

[6] www.eyesweb.org

[7] www.mine-control.com

[8] www.biopus.com.ar

Todas consultadas el 4/Abr/2008 y actualmente en línea.