Índice general

Capítulo 1

Introducción

El análisis de imágenes comprende los procesos que se utilizan con el fin de extraer información de las mismas. Uno de ellos es el de segmentación, que tiene por propósito obtener una representación de la imagen que facilite encontrar los objetos que se encuentran en ella.

Entre los métodos más populares de segmentación de imágenes, se encuentra el denominado Transformación *Watershed*, que procesa una imagen en escala de grises y genera como resultado una en blanco y negro, en la cual se resaltan los contornos de las figuras que se encuentran en la imagen original. Si bien tiene muchas aplicaciones y es de gran utilidad en ciertos casos, el resultado, al ser en blanco y negro, sufre la pérdida de la información brindada por los tonos de grises de la imagen original, lo que puede dar lugar a dificultades al momento de establecer algunas propiedades de la imagen.

Este trabajo focalizará en una variante de la mencionada Transformación Watershed, denominada *Watershed* Topológico. Naturalmente, este enfoque también resalta los contornos de los objetos de la imagen, pero tiene como principal ventaja que mantiene la información de los tonos de grises. Concretamente, se harán tres implementaciones de algoritmos que calculan el Watershed Topológico de una imagen en escala de grises. La primera será una implementación ingenua, con fines principalmente didácticos. La segunda será una de orden cuasi-lineal y, finalmente, la tercera será una concurrente, con el objetivo de mejorar el tiempo de procesamiento.

1.1. Motivación y Objetivos

Este trabajo está dirigido a personas con conocimientos en programación, que deseen introducirse al campo de la Transformación Watershed Topológico. En este informe se describen los conceptos teóricos necesarios para comprender el procedimiento, junto a la correspondiente bibliografia para aquel lector que se interese en profundizar.

Como resultado final, se ofrecerá el código fuente en lenguaje C++ co-

rrespondiente a cada implementación, de modo que el lector pueda apreciar los detalles finos de implementación, como así también hacer modificaciones para ver los efectos en la imagen resultante. Además, se ofrecerán los ejecutables de cada implementación, lo que permitirá al lector hacer pruebas con imágenes de su interés.

Se pretende que el procesamiento de una imagen se realice en un tiempo razonable, por lo que se harán análisis de eficiencia que grafiquen el comportamiento del ejecutable. Por este motivo, también está dedicado a aquellas personas que deseen aplicar la transformación a imágenes propias con fines personales.

Para cumplir con tales fines, será muy importante que el código fuente pueda ser comprendido fácilmente por terceros, como así también que sea eficiente, constituyendo una gran motivación para todo desarrollador de software, como así también una carta de presentación profesional.

1.2. Estado del Arte

Tal como se mencionó previamente, se ha propuesto como objetivo de este trabajo lograr una implementación eficiente. En este sentido, se encuentra descripto en [5] un algoritmo secuencial de orden cuasi lineal para llevar a cabo la transformación, por lo que será una de las bibliografías fundamentales de este trabajo. Cabe mencionar en este punto, que existe una implementación en código abierto, en lenguaje C, que puede ser descargada de `http://perso.esiee.fr/~info/tw/index.html`.

Sin embargo, este algoritmo no saca rédito de los procesadores multi-core de la actualidad, por lo que el foco se pondrá en su paralelización. Respecto a esto, vale la pena mencionar que existe un algoritmo paralelo descripto en [8], aunque en el presente trabajo se introducirá una implementación alternativa.

1.3. Método de medición

Se lanzará los ejecutables en una máquina con las siguiente especificación técnica:

- sistema operativo: Ubuntu 12.04 32-bit
- microprocesador: Intel(R) Core(TM) i3-3120M CPU @ 2.50GHz x 4
- memoria RAM: 4GB

Las imágenes escogidas para hacer las pruebas son las siguientes:

- imagen1

- imagen2
- imagen3

Por cada imagen, se ejecutará la implementación cuasi-lineal y la implementación paralela en 2, 4, 8 y 16 threads y se medirá las siguientes variables:

- consumo de memoria
- porcentaje de CPU
- tiempo de ejecución

1.4. Estructura del Trabajo

En esta sección se pretende indicar cómo está constituído este informe para facilitar la lectura y comprensión. En el mismo, se encuentran, además del actual, cuatro capítulos:

- **Capítulo 2 (Watershed Topológico)**: aquí se introduce la definición de Watershed Topológico. En base a la misma se propone un algoritmo fuerza bruta y se expondrán los resultados de las pruebas de $performance$ correspondientes.
- **Capítulo 3 (Watershed Topológico Cuasi-Lineal)**: en este capítulo se presenta el algoritmo de orden cuasi lineal descripto en [5]. Nuevamente, se expondrán los resultados de las pruebas de $performance$ ya mencionadas.
- **Capítulo 4 (Watershed Topológico en Paralelo)**: en este apartado se propondrá la paralelización del algoritmo propuesto en el capítulo 3, analizando la complejidad que esto supone y, finalmente, presentando los resultados de las pruebas de $performance$ correspondientes.
- **Capítulo 5 (Conclusión)**: como su nombre lo indica, se muestra la apreciación final y los resultados alcanzados de este trabajo.

Además, en este informe se incluyen dos anexos. El primero, Transformación Watershed, introuduce la definición de la transformación. A quienes no la conozcan, es recomendado empezar por aquí, porque, tal como se ha mencionado previamente, la transformación Watershed Topológica es una variante de esta. El segundo, incluye algunas nociones de grafos a conocer para comprender el contenido aquí desarrollado.

Capítulo 2

Watershed Topológico

En este capítulo, se introducirá una variante de Transformación Watershed propuesta por M. Couprie y G. Bertrand [2], denominada Watershed Topológico, que hace foco en la preservación de ciertas características topológicas de la imagen. A continuación se mencionará la definición formal junto a una breve reseña de las propiedades de esta transformación y se propondrá un algoritmo fuerza-bruta que permite calcularla, acompañado de un análisis de su rendimiento.

2.1. Definiciones Preliminares

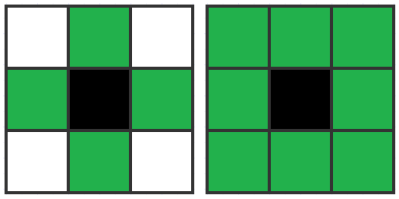
La forma más sencilla de representar una imagen digital en escala de grises, es la denominada representación *matricial*, que consiste en exhibir la imagen como una matriz numérica, donde cada celda corresponde a un pixel. Un ejemplo de ello puede observarse en la Figura 2.1.

2	3	2
2	3	2
2	3	1

Figura 2.1: Imagen digital en escala de grises

Se hace referencia a los píxeles de acuerdo a las coordenadas fila-columna que este ocupa en el arreglo. De esta manera, se habla del pixel (x, y), para referirse al ubicado en la fila x y columna y de la matriz. De este modo, se ve que el pixel $(1, 0)$ de la imagen de la Figura 2.1 tiene nivel de gris 2.

En el ambiente del procesamiento de imágenes, suele ser útil establecer relaciones entre píxeles, conocidas como *relaciones de vecindad*. Las dos más populares son las conocidas como *relacion de vecindad tipo* 4 y



(a) (b)

Figura 2.2: Relación de vecindad entre pixeles. (a) Relación de vecindad tipo 4 (b) Relación de vecindad tipo 8

relacion de vecindad tipo 8. Para un pixel (x, y), la primera relación está constituída por los píxeles $(x-1, y), (x+1, y), (x, y-1), (x, y+1)$, mientras que la restante, por los píxeles $(x-1, y-1), (x-1, y), (x-1, y+1), (x, y-1), (x, y+1), (x+1,$

Ambas son descriptas en la imagen 2.2

Una representación de imagen digital alternativa, adecuada para expresar las relaciones de vecindad, es la conocida como *Grilla Digital*, que consiste en exhibir la imagen como un grafo con pesos conectado y simétrico (E, Γ, F) [1], donde $E \subseteq \mathbb{Z}^2$ es el conjunto de todos los pixeles de la imagen, definidos por sus coordenadas fila - columna, Γ describe la *relacion de vecindad* entre ellos y $F : E \to \mathbb{Z}$, de modo que para cada pixel $p \in E$, $F(p)$ indica el nivel de gris de p. En la Figura 2.3 podemos observar esta representación para la imagen de la Figura 2.1, tomando una relación de vecindad tipo 4.

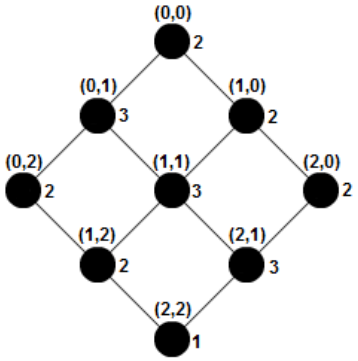


Figura 2.3: Grilla Digital de la imagen de la Figura 2.1 para relación de vecindad tipo 4. Arriba de cada nodo se observa su nombre, y al costado su peso

A continuación se presentan las definiciones que serán utilizadas en el desarrollo del presente capítulo.

Definición: Sea (E, Γ, F) una grilla digital, entonces llamaremos k-sección superior, o sección superior de nivel k, al conjunto de nodos cuyo

[1] Para repasar nociones de grafos ver Anexo B. Grafos

peso es mayor o igual a k. Formalmente

$$F_k = \{x \in E; F(x) \geq k\}$$

En la Figura 2.4 podemos observar, resaltados en verde, los nodos que conforman la F_2 de la grilla digital de la Figura 2.3.

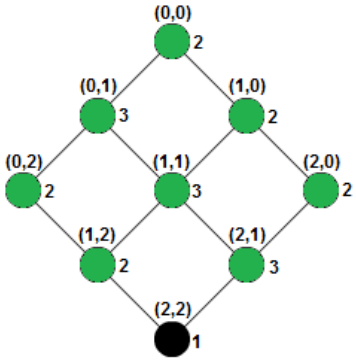


Figura 2.4: F_2 de la grilla digital de la Figura 2.3

Definición: Sea (E, Γ, F) una grilla digital, entonces llamaremos k-sección inferior, o sección inferior de nivel k, al conjunto de nodos cuyo peso sea menor a k. Formalmente

$$\overline{F_k} = \{x \in E; F(x) < k\}$$

En la Figura 2.5 podemos observar, resaltados en verde, los nodos que constituyen la $\overline{F_3}$ de la grilla digital de la Figura 2.3.

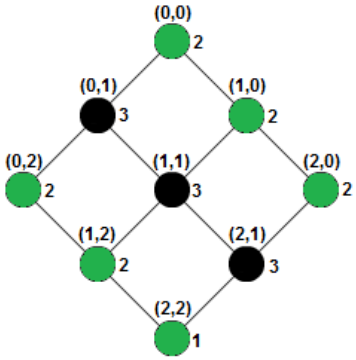


Figura 2.5: $\overline{F_3}$ de la grilla digital

Definición: Sea (E, Γ, F) una grilla digital, entonces llamaremos k-componente superior, o componente superior de nivel k, a una componente conectada de F_k.

Por ejemplo, a partir de F_2, descripta en la Figura 2.4, podemos concluir que esta grilla digital contiene una sola 2-componente superior, compuesta por todos los nodos de F_2.

Definición: Sea (E, Γ, F) una grilla digital, entonces llamaremos k-componente inferior, o componente inferior de nivel k, a una componente conectada de $\overline{F_k}$.

Para ilustrar esta definición, podemos tomar $\overline{F_3}$ de la Figura 2.5, veremos que hay dos 3-componente inferior, una compuesta por los nodos $(0,0),(1,0),(2,0)$ y la otra por $(0,2),(1,2),(2,2)$.

Definición: Sea (E, Γ, F) una grilla digital, entonces llamaremos mínimo regional de F a una k-componente inferior que no contenga una $(k-1)$-componente inferior.

Para el caso de la grilla de la Figura 2.3, si calculáramos todas las componentes inferiores, concluiríamos que existen dos mínimos regionales, uno compuesto por los nodos $(0,0),(1,0),(2,0)$ y la otra por el nodo $2,2$, lo cual concuerda con la noción intuitiva de mínimos regionales dada en el Anexo A.

2.2. Definición de Watershed Topológico

La definición formal de Watershed Topológico de una imagen, se basa en el concepto de nodo W-simple. Sea (E, Γ, F) una grilla digital y $X \subset E$ no vacío, entonces un nodo $x \in X$ es W-simple para X si es adyacente a exactamente una componente conectada de $\overline{X}$.

Como ejemplo, se puede tomar la grilla de la Figura 2.3 y tomar $X = F_3 = \{(0,1),(1,1),(2,1)\}$. Tal como se ha mencionado previamente y como puede verse fácilmente en la Figura 2.5, $\overline{F_3}$ contiene dos componentes conectadas, una compuesta por los nodos $(0,0),(1,0),(2,0)$ y la otra por $(0,2),(1,2),(2,2)$, por lo tanto, ninguno de los nodos de F_3 es W-simple, pues todos ellos son adyancentes a las dos componentes conectadas de $\overline{F_3}$.

Ahora bien, sea (E, Γ, F) una grilla digital, sea $p \in E, k = F(p)$, entonces, decimos que el nodo p es **W-destructuble** para F si es W-simple para F_k. Decimos que $G : E \to \mathbb{Z}$ es *Watershed* Topológico de F, si G se obtiene a partir de F disminuyendo sucesivamente el valor de puntos W-destructible en uno hasta lograr estabilidad, es decir, que no queden puntos W-destructible. En la nomenclatura de Transformaciones Watershed, las *Catchment Basins* de G serán los mínimos regionales de G y el resto de los puntos serán las *Watershed Lines*.

Continuando con el ejmplo de la Figura 2.3, es sencillo ver que el nodo $(1,2)$ es W-destructible, pues $F((1,2)) = 2$ y al observar que $F_2 = E - (2,2)$, y advertir que F_2 constituye una componente conectada, queda claro que $\overline{F_2} = (2,2)$ es adyacente a tal componente conectada y, por ende, es W-destructible.

2.3. Bondades

Si bien la Transformación Watershed de imágenes en escala de grises constituye una herramienta de gran utilidad y un paso muy importante en la segmentación de imágenes, la mayoría de los enfoques existentes, tal como se menciona en [4], tienen algunas desventajas. En primera instancia, el resultado es una imagen binaria, lo que implica que se pierde la información brindada por los tonos de grises de la imagen original, lo que limita su utilización para procesamiento adicional. Además, muchos de ellos, como el basado en el paradigma de inundación, producen Watershed Lines que no necesariamente se encuentran en los contornos más significativos de la imagen original. Esto significa que pueden presentarse serias dificultades al intentar obtener conclusiones a partir de la imagen resultante.

En contrapartida, la transformación presentada en este capítulo preserva la conectividad de cada $\overline{F_k}$, es decir, la cantidad de componentes conectadas es la misma tanto en la imagen original como en su transformación, en cada $\overline{F_k}$. Esto constituye una gran virtud, y las implicaciones de la misma se encuentran detalladas en [3].

2.4. Implementación Fuerza Bruta

En este apartado se presentará un algoritmo para llevar a cabo la Transformación Watershed Topológico de una imagen. La idea será recorrer la grilla nodo por nodo, verificando si cada uno es W-destructible o no. En caso afirmativo se disminuirá su nivel de gris en 1, caso contrario su valor no cambiará. Una vez escaneada la imagen en su totalidad, si se encontró algún nodo W-destructible, se repite todo el proceso. Caso contrario, el algoritmo termina.

Es sencillo ver que la dificultad del algoritmo se encuentra en cómo determinar si un nodo es W-destructible o no, asique se comenzará por estudiar esta situación. Sea (E, Γ, F) una grilla y sea $p \in E$ y $k = F(p)$, una observación clave es que p será W-destructible si y sólo si cada uno de los $q \in \Gamma^-(p)$, donde $\Gamma^-(p) = \{n \in \Gamma(p) : F(n) < F(p)\}$, pertenecen a la misma componente conectada en $\overline{F_k}$. A partir de esto, podemos distinguir dos casos triviales, uno es aquel en el que $\Gamma^-(p) = \emptyset$, en este caso p no será W-destructible puesto que no será adyacente a ninguna componente conectada en $\overline{F_k}$. El segundo, es la situación en la que $\Gamma^-(p)$ está compuesto por sólo un $q \in E$, en este caso p será W-destructible puesto que es adyacente sólo a la componente conectada a la que pertenece q en $\overline{F_k}$.

Con esto en mente, sólo queda por resolver el caso en el que p tenga más de un vecino con nivel de gris inferior a k. La idea será calculár cada uno de los nodos alcanzables a partir de algún $q \in \Gamma^-(p)$ en $\overline{F_k}$, y luego chequear si $\Gamma^-(p)$ está contenido en dicho conjunto.

A continuación se presenta el pseudocódigo para determinar si un $p \in E$ es W-destructible o no.

Algorithm 1

```
1:  function isWdestructible(E, Γ, F, p)
2:      if Γ⁻(p) = ∅ then return false
3:      if Γ⁻(p) · size = 1 then return true
4:      reachableNodes ← ∅
5:      index ← 0
6:      reachableNodes · insert(Γ⁻(p)[0])
7:      while ¬(reachableNodes · containAll(Γ⁻(p))) ∧ index < reachableNodes · size do
8:          q ← reachableNodes[index]
9:          for all n ∈ Γ(q) do
10:             if F(n) < F(p) then
11:                 reachableNodes · insert(n)
12:         index ← index + 1
        return reachableNodes · containAll(Γ⁻(p))
```

Se recibe por parámetro un grafo con pesos y un $p \in E$, el resultado será un valor booleano indicando si p es W-destructible o no. Las líneas 2 y 3 reflejan los casos triviales mencionados anteriormente. De la línea 4 a la 6 se inicializan dos variables locales, entre ellas reachableNodes, un set (por definición no admite valores duplicados) con sólo un elemento, algún $q \in \Gamma^-(p)$, que llamaremos *semilla*. A partir de la línea 7 comienza un loop que insertará en reachableNodes aquellos nodos alcanzables a partir de la semilla en $\overline{F_k}$. En la primera iteración se insertarán aquellos $n \in \Gamma(q)$ tales que $F(n) < F(p)$, es decir, $n \in \overline{F_k}$. La siguiente iteración tomará el siguiente elemento del set y se repetirá el proceso. Se iterará hasta que, o bien $\Gamma^-(p) \subseteq reachableNodes$, esto significa que todos los $q \in \Gamma^-(p)$ pertencen a la misma componente conectada de $\overline{F_k}$, o bien hayamos calculado todos los $t \in E$ alcanzables a partir de la semilla en $\overline{F_k}$, con lo que concluimos los contrario, y por lo tanto p no es W-destructible.

Para llevar a cabo la transformación, sólo es necesaria una rutina que itere cada uno de los $p \in E$ en busca de puntos W-destructubles, disminuya su valor si corresponde, y repita el proceso hasta lograr estabilidad.

A continuación se presenta el pseudocódigo del algoritmo que lleva a cabo la transformación.

2.5. Conclusión

Pendiente

Algorithm 2

```
1: procedure doTopologicalWatershed(E, Γ, F)
2:     repeat
3:         someChange ← false
4:         for all p ∈ E do
5:             if isWdestructible(E, Γ, F, p) then
6:                 F(p) ← F(p) − 1
7:                 someChange ← true
8:     until someChange = false
```

Capítulo 3

Watershed Topológico Cuasi-Lineal

En [5] se propone un algoritmo de orden cuasi-lineal para el cálculo del Watershed Topológio de una imagen. Se logra esta notable eficiencia a partir de dos factores. Por un lado, los pixeles se procesan siguiendo un orden de modo de asegurar que su valor se reducirá, a lo sumo, sólo una vez durante toda la ejecución del algoritmo. Por el otro, se disminuye el valor de un pixel W-destructible en el mínimo valor posible, en lugar de disminuirlo sólo en 1. Esto se logra gracias a la utilización de una estructura de datos llamada "Árbol de Componentes", que será introducida a continuación.

En este capítulo, repasaremos en primer lugar algunas definiciones básicas, luego se introducirá el concepto de Árbol de Componentes de una imagen, para después explicar el rol del mismo en el cálculo del Watershed Topológico. Finalmente se dará un pseudo-código junto a una explicación línea a línea, a fin de mostrar el procedimiento detalladamente.

3.1. Definiciones preliminares

A continuación se presentan conceptos clave que se utlizarán en el desarrollo de este capítulo. Sea $F \in \mathcal{F}, k \in \mathbb{K}$, denotaremos con $C_k(F)$ el conjunto de todas las k-componentes superiores de F, y denotaremos con C(F) el conjunto de todas las componentes superiores de F, es decir

$$C(F) = \bigcup_{k=k_{min}}^{k_{max}} C_k(F)$$

Análogamente definimos $C_k(\overline{F})$ y $C(\overline{F})$.

Para seguir con el ejemplo del capítulo anterior de la Figura **??**, podemos ver que $C_2(F)$ está compuesto por CS2 y CS3, mientras que C(F) está

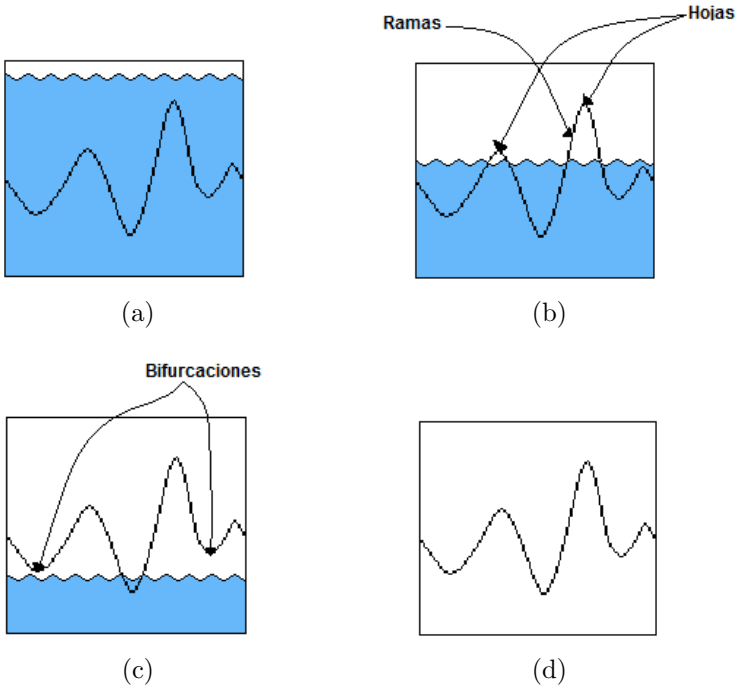


Figura 3.1: Árbol de componentes. (a) Inicio (b) y (c) Imagen parcialmente inundada (d) Fin

compuesta por CS1,CS2, .. , CS12. Del mismo modo, $C_2(\overline{F})$ esta compuesta por CI1, CI2, .. CI5, y $C(\overline{F})$, por CI1, .. , CI10.

3.2. Árbol de Componentes

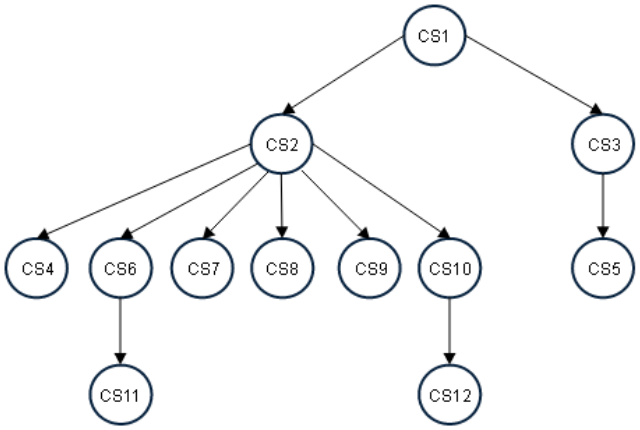
Se piensa una imagen como un relieve topográfico completamente sumergido en agua, al que se le hacen perforaciones en sus mínimos. El nivel de agua disminuirá lentamente, con lo que se comenzará a observar islas (correspondientes a los máximos de la imagen), estas conformarán las hojas del árbol de componentes de la imagen. A medida que el nivel de agua continue descendiendo, las islas crecerán en tamaño, conformando lo que serán las ramas del árbol. Luego, a partir de algún nivel, algunas islas se fusionarán conformando una sola pieza, estas piezas serán las bifurcaciones del árbol. El proceso terminará cuando toda el agua haya desaparecido. En la Figura 3.1 podemos observar el proceso descripto.

Lo que acabamos de ver es una definición intuitiva de qué es el Árbol de Componentes de una imagen, de modo de facilitar la comprensión del lector. A continuación la definición formal.

Sea $F \in \mathcal{F}$ y $k, j \in \mathbb{K}$. Entonces definimos el **Árbol de Componentes**

	0	1	2	3	4
0	CS4	CS2	CS2	CS1	CS5
1	CS1	CS2	CS6	CS6	CS1
2	CS2	CS7	CS2	CS11	CS2
3	CS8	CS2	CS1	CS1	CS1
4	CS1	CS9	CS2	CS10	CS12

(a)



(b)

Figura 3.2: (a) Mapeo de Componentes, (b) Árbol de Componentes. En ambos casos respecto a la imagen de la Figura **??** (a)

de F, T(F), como un árbol cuyos nodos serán elementos de C(F) y habrá una arista de $c' \in C_k(F)$ hacia $c \in C_j(F)$ si j = k+1 y $c \subseteq c'$. En este caso, decimos que c' es padre de c, y también que c es hijo de c'.

El **mapeo de componentes** será un mapa $\Psi : E \to C(F)$, que relaciona cada $p \in E$ con el elemento de C(F) que lo contiene.

Algo muy importante acerca de esta estructura de datos, es que existe un algoritmo de orden cuasi-lineal para su cálculo.[1]

Continuando con el ejemplo del capítulo anterior, en la Figura 3.2 vemos el mapeo de componentes junto al Árbol de componentes correspondiente.

A continuación se presenta la definición de $highestfork$ del árbol, el cual, como se verá más adelante, es fundamental para el cálculo de Watershed Topológico de una imagen. Para hacer más sencilla la lectura, vamos a denotar un nodo de T(F) como un par $[k, c]$ cuando $c \in C_k(F)$ y llamaremos nivel del nodo al número k.

[1]Para más información consultar [6]

Sea $F \in \mathcal{F}$ y $[k,c],[k_1,c_1],[k_2,c_2],....,[k_n,c_n] \in C(F)$, entonces

- $[k_1,c_1]$ es **ancestro** de $[k_2,c_2]$ si $k2 \leq k1$ y $c2 \subseteq c1$. En este caso, también decimos que $[k_1,c_1]$ está encima de $[k_2,c_2]$ y que $[k_2,c_2]$ está abajo de $[k_1,c_1]$.
- $[k,c]$ es **ancestro común** de $[k_1,c_1]$,$[k_2,c_2]$ si $[k,c]$ es ancestro de $[k_1,c_1]$ y $[k,c]$ es ancestro de $[k_2,c_2]$.
- $[k,c]$ es el **menor ancestro comun** de $[k_1,c_1]$,$[k_2,c_2]$, si $[k,c]$ es ancestro de comun de ambos y no hay ningún otro ancestro común abajo de $[k,c]$.
- $[k,c]$ es el **menor ancestro común propio** de $[k_1,c_1]$,$[k_2,c_2]$ si $[k,c]$ es ancestro común de ambos y $[k,c]$ es distinto de $[k_1,c_1]$ y $[k,c]$ es distinto de $[k_2,c_2]$.
- $[k_1,c_1]$,$[k_2,c_2]$ están **separados** si tienen menor ancestro común propio, caso contrario decimos que están linkeados.
- $[k,c]$ es **highest fork** de M = {$[k_1,c_1]$,$[k_2,c_2]$,.....,$[k_n,c_n]$} si las siguientes condiciones se satisfacen:
  - Si dos nodos $[k_i,c_i]$,$[k_j,c_j]$ están separados, entonces el menor ancestro común tiene altura $\leq k$.
  - Existen nodos $[k_i,c_i]$,$[k_j,c_j]$ separados, tal que $[k,c]$ es el menor ancestro común propio de $[k_i,c_i]$,$[k_j,c_j]$.

Se tomará como ejemplo el Árbol de la Figura 3.2, se puede ver que el conjunto {CS2,CS6,CS11} no tiene highest fork, pues no existen nodos separados. Sin embargo CS2 es el highest fork de {CS2,CS4,CS7}, pues CS4 y CS7 están separados. Es importante notar que el highest fork de un conjunto puede no pertenecer a ese conjunto, por ejemplo si se toma {CS2,CS3} su highest fork es CS1.

3.3. Algoritmo

En este apartado se explicará cómo utilizar el Árbol de Componentes en el cáclulo de la transformación.

Sea $F \in \mathcal{F}$, para calcular el Watershed Topológico de F vamos a calcular $T(\overline{F})$ teniendo en cuenta que $\overline{F}(p) = \mathrm{F(p)} + 1$ [2]. En este caso, habrá una arista de $c' \in C_k(\overline{F})$ hacia la componente $c \in C_j(\overline{F})$ si j = k+1 y $c' \subseteq c$.

A continuación se menciona un resultado que da lugar al algoritmo propuesto, el lector interesado en la demostración puede consultar [5].

[2]Esto se desprende del concepto de stacks, para más información ver [2]

Sea $F \in \mathcal{F}$, sea $p \in E$. Denotamos como $V(p) = \{[k, C(q)] : q \in \Gamma^-(p)\}$, donde $k = \overline{F}(q)$ y $C(q)$ será la componente $c \in C_k(\overline{F})$ que contiene a q.

- Si V(p) $\neq \emptyset$ y V(p) no tiene highest fork en $T(\overline{F})$, entonces p será W-destructible para F para todos los valores $k : w \leq k \leq F(p)$, y no será W-destructible para F para valor w - 1, donde w es el mínimo nivel de V(p).
- Si V(p) $\neq \emptyset$ y V(p) tiene highest fork en $T(\overline{F})$ cuyo nivel es $w \leq$ F(p), entonces p será W-destructible para F para todos los valores $k : w \leq k \leq F(p)$, y no será W-destructible para F para w - 1.

Este resultado es de gran importancia, pues permite reducir las iteraciones del algoritmo fuerza bruta en gran medida. Claro, en lugar de disminuir cada punto W-destructible en 1 y luego iterar toda la imagen nuevamente, en cada iteración se logran avances sustanciales para eliminar los puntos W-destructibles de la imagen.

Tal como se mencionó en la reseña del presente capítulo, existe una forma de asegurar que el valor de cada punto se disminuya, a lo sumo, sólo una vez. Es una técnica muy sencilla, y consiste disminuir en primero aquellos puntos W-destructibles cuyo valor resultante (determinado por el resultado anterior) sea menor.

3.4. Implementación

La observación clave para llevar a cabo un algoritmo, es que al disminuir el valor de un $p \in E$ de acuerdo al resultado visto anteriormente, $\Psi(p)$ pasará a ser o bien el mínimo de V(p), o bien el highest fork de V(p), de acuerdo a si existe el highest fork o no. Esto nos permitirá mantener actualizado el mapeo de componentes a medida que se disminuyan los valores de cada uno de los puntos W-destructibles.

En base a esta observación, se definirça la función $isWdestructible$ de modo que retorne un nodo. De manera que, si la función retorna $[k, c]$ para un $p \in E$, entonces el valor de p debe disminuirse a (k - 1), y una vez hecho esto $\Psi(p) = [k, c]$.

A continuación se muestra el pseudo código de esta función.

Es importante mencionar que en [5], se establece cómo calcular el highest fork de un set $s \subseteq C(\overline{F})$ a partir del cálculo del menor ancestro común de dos elementos, para lo cual existe un algoritmo sencillo y eficiente detallado en [7].

Ahora se detallará el algoritmo que lleva a cabo la transformación. Se utilizará un mapa : $E \to C(\overline{F})$, que asocia cada $p \in E$ con el nodo retornado por la función isWdestructible. Para procesar cada uno de los $p \in E$ en el orden conveniente, lo único que se necesitará es que ese mapa este ordenado de modo creciente con respecto al nivel del nodo.

Algorithm 3

```
1: function isWdestructible(E, F, Γ, p, T(F̄), Ψ)
2:     V ← ∅
3:     for all q ∈ Γ⁻(p) do
4:         V · insert(Ψ(q))
5:     if V = ∅ then return null
6:     [k, c] ← highestFork(T(F̄), V)
7:     if [k, c] = null then return minLevel(V)
8:     if k ≤ F(p) then return [k, c]
       return null
```

A continuación, se explicará la idea del algoritmo. Lo primero que debe hacer es popular el mapa con aquellos $p \in E$ W-Destructibles. Luego, debemos iterar el mapa y extraer cada uno de los elementos. En las sucesivas iteraciones se obtiene algún $p \in E$, se disminuye su valor y se actualiza el mapeo de componentes. Puesto que el valor de p ha cambiado, hay que recalcular el estado de cada uno de los vecinos de p, para lo que se volverá a utilizar la función isWdestrucible, y se actualizará el mapa en consecuencia.

A continuación se da el pseudocódigo que lleva a cabo la transformación de acuerdo al proceso descripto.

Algorithm 4

```
1: procedure doTopologicalWatershed(E, Γ, F, T(F̄), Ψ)
2:     componentMap ← ∅
3:     for all p ∈ E do
4:         [i, c] ← isWdestructible(E, F, Γ, p, T(F̄), Ψ)
5:         if ¬[i, c] = null then
6:             componentMap · insert(p, [i, c])                ▷ sorted by i
7:     while ¬componentMap · empty() do
8:         p, [i, c] ← componentMap · extractFirst()
9:         F(p) ← i − 1
10:        Ψ(p) ← [i, c]
11:        for all q ∈ Γ(p), F(q) ≥ i do    ▷ if F(q) < i, then q has already
       been processed so can not be W-destructible
12:            [j, d] ← isWdestructible(E, F, Γ, q, T(F̄), Ψ)
13:            if [j, d] = null then
14:                componentMap · removeIfExists(q)
15:            else
16:                componentMap · addOrUpdate(q, [j, d])
```

3.5. Conclusión

Pendiente

Capítulo 4

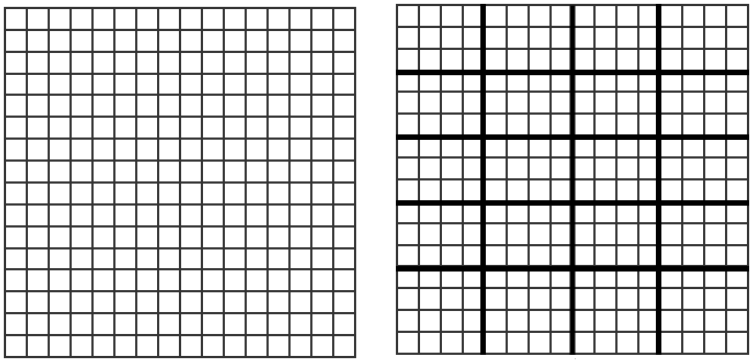
Watershed Topológico en Paralelo

En este capítulo, vamos a presentar un algoritmo paralelo para el cálculo del Watershed Topológico de una imagen, basado en el algoritmo introducido en el capítulo anterior. Para ello, se hará un análisis de las dificultades que introduce la paralelización y se propondrá un algoritmo acorde. Finalmente, se brindará un pseudocódigo que implemente la solución propuesta, junto a un análisis de su rendimiento en términos de eficiencia.

4.1. Presentación del problema

Básicamente, el algoritmo cuasi-lineal descripto en el capítulo anterior, se vale del árbol de componentes de la imagen para hacer la transformación, y puede resumirse en dos pasos: el primero, scannear la imagen en su totalidad y armar un mapa ordenado que indique, para cada punto W-destructible, el valor al que debe disminuirse y la componente a la que pertenecerá cuando esto suceda, el segundo, extraer cada punto de ese mapa, disminuir su valor, actualizar el árbol de componentes y, finalmente, actualizar el mapa recalculando la W-destructibilidad de cada uno de sus vecinos. El algoritmo termina una vez que el mapa se vacía.

La idea del algoritmo en paralelo que se propone en este trabajo, se basa en dividir en fragmentos la imagen a transformar y aplicar, en cada uno, el algoritmo cuasi-lineal de manera concurrente, tal como si fuesen imágenes distintas. Notar que no hay sólo una forma de dividir la imagen, por ello, vale aclarar que se dividirá de acuerdo al divisor más cercano a la raiz cuadrada del número de fragmentos deseados, así por ejemplo si se desea dividir la imagen en 20 fragmentos, se dividirá la imagen en 4 fragmentos iguales (salvo quizás el último), y cada subfragmento en 5 subfragmentos iguales (salvo quizás el último). Esta división puede verse gráficamente en la Figura 4.1.



(a) (b)

Figura 4.1: (a) Imagen original. (b) División en 20 fragmentos.

El primer paso mencionado previamente, puede hacerse en paralelo sin problemas puesto que la imagen sólo se lee y, por lo tanto, no hay problemas de sincronización. Respecto del segundo, surgen problemas de paralizaciòn evidentes. Una situación que lo evidencia es la siguiente: se supone un thread t corriendo en el fragmento A de la imagen, mientras, concurrentemente, otros threads corren en cada uno de los otros fragmentos. Sistemáticamente, lo que hace t es extraer algún punto W-destructible p ($\in$ fragmento A), disminuir su valor, actualizar el árbol y actualizar el mapa para cada uno de los $q \in \Gamma(p)$. ¿Qué sucede si algún q pertenece a otro fragmento? El valor de q puede haber cambiado respecto del momento en el que se armó el mapa que utiliza t, y, en consecuencia, este puede haber quedado desactualizado, con lo cual no se puede estar seguro del valor al que debe ser disminuido p. Del mismo modo, si se dismunuyera p, quedarán desactualizados los mapas de los fragmentos en los que p tenga vecinos. Debido a estos problemas, se terminará con una imagen resultante que no satisface la definición de Watershed Topológico.

De aquí en adelante, se dirá que un punto es *borde* si posee vecinos en fragmentos ajenos, desde luego esto dependerá de la relación de vecindad que se esté utilizando. En la Figura 4.2 se observan los puntos bordes de una imagen dividida en 4 fragmentos, tomando una relación de vecindad tipo 4 o tipo 8.

4.2. Estado del arte

Como se vio en el apartado anterior, el problema para hacer un algoritmo paralelo puede resumirse en contestar la pregunta ¿Cómo resolver los problemas de sincronización que presentan los puntos borde?

Una de las respuestas que primero viene a la cabeza, es que todos los

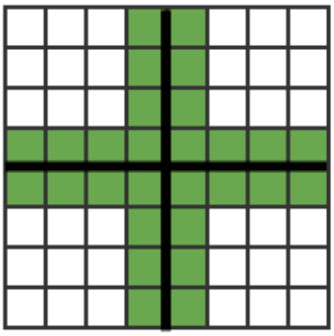


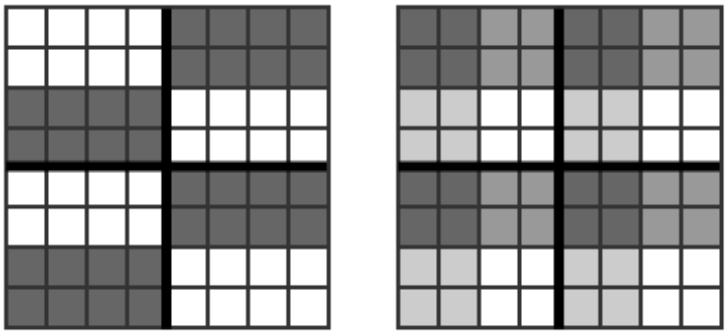
Figura 4.2: En verde se muestran los puntos borde de una imagen dividida en 4 fragmentos, tomando una relación de vecindad tipo 4 o tipo 8.

threads utilicen un mapa global y que cada thread actualice el valor de los puntos que pertenecen a su fragmento y de los vecinos de los puntos borde pertenecientes a fragmentos ajenos. Sin embargo, el problema de sincronización sigue existiendo, puesto que puede modificarse al mismo tiempo el valor de un punto borde por un thread, y el valor de su vecino en otro fragmento por otro thread, con lo cual no podemos asegurar que la transformación sea correcta.

En [8] se propone una solución basada en particionar cada fragmento en subfragmentos, de modo de asegurar que si un thread está modificando un punto, ningún otro esté modificando algún vecino perteneciente a otro fragmento. De este modo, cada thread procesa un subfragmento i, y, cuando todos terminan, cada uno procesa el $i+1$, continuando de esta manera hasta procesar todos los subfragmentos.

Por supuesto, una sola iteración puede ser insuficiente puesto que al disminuir un punto p que pertenece a un subfragmento, puede que alguno de sus vecinos en algun otro subfragmento se vuelva W-destructible. Por este motivo, se utiliza un arreglo global para marcar los puntos que hayan sido modificados, y asi procesar nuevamente cada uno de los vecinos en su respectivo subfragmento.

La desventaja que presenta este enfoque, es que el modo de particionar los fragmentos depende del tipo de vecindad que se utilice, tal como puede observarse en la Figura 4.3, lo que supone un gran problema de escalabilidad del algoritmo, puesto que para cada relación de vecindad que desee soportar el algoritmo, habrá que idear un subfragmentado acorde (que puede no ser trivial).



(a) (b)

Figura 4.3: Subfragmentación para una imagen dividida en 4. (a) Subfragmentación para el caso de una relación de vecindad de tipo 4. Cada fragmento se divide en 2 subfragmentos. (b) Subfragmentación para el caso de una relación de vecindad de tipo 8. Cada fragmento se divide en 4 subfragmentos.

4.3. Algoritmo

La solución que se presenta en este trabajo, consiste en tratar los puntos borde de modo secuencial, y, de este modo, superar la desventaja del algoritmo descripto anteriormente.

De modo concreto, se propone dividir la imagen en fragmentos y, de forma concurrente, calcular el mapa para cada uno tal como lo hace el algoritmo cuasi-lineal. Luego, utilizar el mapa de modo análogo al algoritmo cuasi-lineal, con la salvedad de, en caso de haber extraído del mapa un punto borde, este se marca en un arreglo global y no se modifica su valor.

Una vez que todos los threads terminan, un único thread repite el proceso para los puntos bordes marcados en el arreglo global, marcando en un nuevo arreglo aquellos puntos que no sean borde y que deban ser recalculados.

Una vez hecho esto, el proceso se repite nuevamente, donde cada thread se encarga de procesar aquellos puntos marcados en el arreglo global que pertenecen a su fragmento, y así hasta eliminar todos los puntos W-destructibles.

4.4. Implementación

A continuación se presentará el pseudo código del algoritmo propuesto en el apartado anterior.

La siguiente función se encarga de inicializar el mapa de un fragmento pasado como parámetro.

Algorithm 5

```
1: function initializeMap(E, Γ, F, T(F̄), Ψ, Tile)
2:     for all p ∈ Tile do
3:         [i, c] ← isWdestructible(E, F, Γ, p, T(F̄), Ψ)
4:         if [i, c] ≠ null then
5:             componentMap · insert(p, [i, c])          ▷ sorted by i
       return componentMap
```

Notar que coincide con la lógica utilizada por el algoritmo cuasi-lineal, salvo que, en lugar de recorrer toda la imagen, se restringe al set de puntos $Tile$ pasado como parámetro.

Lo que sigue es el procedimiento que se encarga de procesar un fragmento de la imagen.

Algorithm 6

```
1: procedure doTopologicalWatershedOnTile(E, Γ, F, T(F̄), Ψ, Tile)
2:     componentMap ← InitializeQueue(E, Γ, F, T(F̄), Ψ, Tile)
3:     while ¬componentMap · empty() do
4:         p, [i, c] ← componentMap · extractFirst()
5:         if Γ(p) ⊈ Tile then                          ▷ means a border px
6:             borders · insert(p)
7:         else
8:             F(p) ← i − 1
9:             Ψ(p) ← [i, c]
10:            for all q ∈ Γ(p), F(q) ≥ i do
11:                [j, d] ← isWdestructible(E, F, Γ, q, T(F̄), Ψ)
12:                if [j, d] = null then
13:                    componentMap · removeIfExists(q)
14:                else
15:                    componentMap · addOrUpdate(q, [j, d])
```

Nuevamente, notar el parecido al algoritmo cuasi-lineal, lo único que se agrega es el chequeo de si el punto es borde, y en este caso, se extrae del mapa y se marca en el arreglo global borders.

A continuación se muestra la lógica del procedimiento encargado de procesar los puntos borde.

Algorithm 7

```
1: procedure doTopologicalWatershedOnBorder(E, Γ, F, T(F̄), Ψ)
2:     componentMap ← InitializeQueue(E, Γ, F, T(F̄), Ψ, borders)
3:     while ¬componentMap · empty() do
4:         p, [i, c] ← componentMap · extractFirst()
5:         F(p) ← i − 1
6:         Ψ(p) ← [i, c]
7:         for all q ∈ Γ(p), F(q) ≥ i do
8:             [j, d] ← isWdestructible(E, F, Γ, q, T(F̄), Ψ)
9:             if [j, d] = null then
10:                componentMap · removeIfExists(q)
11:                changed · removeIfExists(q)
12:            else
13:                if q ∈ componentMap then
14:                    componentMap · update(q, [j, d])
15:                else
16:                    changed · insert(q)
```

Como puede observarse, es aquí donde se disminuye el valor de los puntos borde, motivo por el cuál esta lógica debe correr en modo secuencial. Al terminar este procedimiento, todos los puntos borde habrán dejado de ser W-destructibles, pero al disminuir su valor, puede que algunos puntos no borde se hayan vuelto W-destructibles, por lo cuál se los marca en el arreglo globlal changed.

Finalmente, aquí se presenta el procedimiento encargado de hacer la transformación Watershed Paralelo.

Algorithm 8

1: **procedure** doTopologicalWatershed(E, Γ, F, $T(\overline{F})$, Ψ, n)
2: $tiles \leftarrow divideImageInTiles(n)$
3: $threadPool \leftarrow createThreadPool(n)$
4: $borders \leftarrow \emptyset$ ▷ global for all threads
5: $changed \leftarrow \emptyset$ ▷ global for all threads
6: **for** $i = 1 \rightarrow n$ **do**
7: $threadPool[i] \cdot run(doTopologicalWatershedOnTile(E, \Gamma, F, T(\overline{F}), \Psi, tiles[i]))$
8: $waitForAllThreadsToFinish()$
9: $doTopologicalWatershedOnBorder(E, \Gamma, F, T(\overline{F}), \Psi)$
10: **while** $changed \neq \emptyset$ **do**
11: **for** $i = 1 \rightarrow n$ **do**
12: $threadPool[i] \cdot run(doTopologicalWatershedOnTile(E, \Gamma, F, T(\overline{F}), \Psi, tiles[i] \cap changed))$
13: $waitForAllThreadsToFinish()$
14: $doTopologicalWatershedOnBorder(E, \Gamma, F, T(\overline{F}), \Psi)$

Notar que se recibe por parámetro la cantidad de threads que se utilizarán en el proceso. En el for de las líneas 6 y 7, cada thread se encarga de procesar su respectivo fragmento, y una vez que todos terminan, se procesan los puntos borde. Luego, mientras haya puntos no bordes por revisar, cada thread procesa los que se encuentran en su fragmento, marcando en el arreglo los puntos borde si los hubiere, y nuevamente se procesan los mismos, repitiendo este proceso hasta lograr estabilidad.

4.5. Conclusión

Pendiente

Capítulo 5

Conclusión

Pendiente

Anexo

Anexo A

Transformación Watershed

En el área de estudio de procesamiento de imágenes, las imágenes en escala de grises comúnmente son representadas como relieves topográficos, esto es, el nivel de gris de un punto representa la altura del mismo en un eje cartesiano 3D. Esta representación favorece la apreciación del efecto de ciertas transformaciones sobre la imagen de estudio.

La segmentación de imágenes, comprende la partición de una imagen de modo de facilitar la distinción de los objetos que hay en la misma respecto del fondo, a partir del resaltado de contornos. Dentro del campo de la morfología matemática, la Transformación Watershed constituye una de las clásicas herramientas para este propósito.

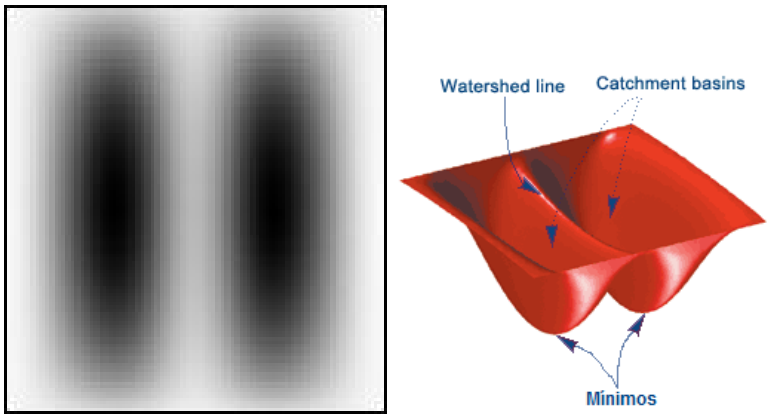
La Transformación Watershed utiliza la representación topográfica de la imagen haciendo foco en 3 tipos de puntos:

- Los mínimos regionales, son aquellos puntos que no pueden alcanzar a un punto más bajo valor sin pasar antes por uno más alto.
- Aquellos puntos en los que, si cayera una gota de agua, indefectiblemente esa gota descendería hacia un mínimo regional determinado. Técnicamente, constituyen las *Catchment Basins*.
- Aquellos puntos en los que, en caso de caer una gota de agua, no podría determinarse hacia qué mínimo descendería. Estos, son los que forman las *Watershed Lines*.

En la Figura A.1 se ejemplifican puntos pertenecientes a cada tipo.

El objetivo de la transformación es particionar la imagen de acuerdo a las watershed Lines.

Uno de los más populares algoritmos para llevar a cabo este procesamiento es el conocido como "Watershed por Inmersión", desarrollado en [1], el cuál se basa en la idea de un relieve topográfico al que se le hacen perforaciones en sus mínimos y, lentamente, se lo sumerge en agua. El nivel de agua cubrirá primero los mínimos y, progresivamente, inundará las *cuencas* del

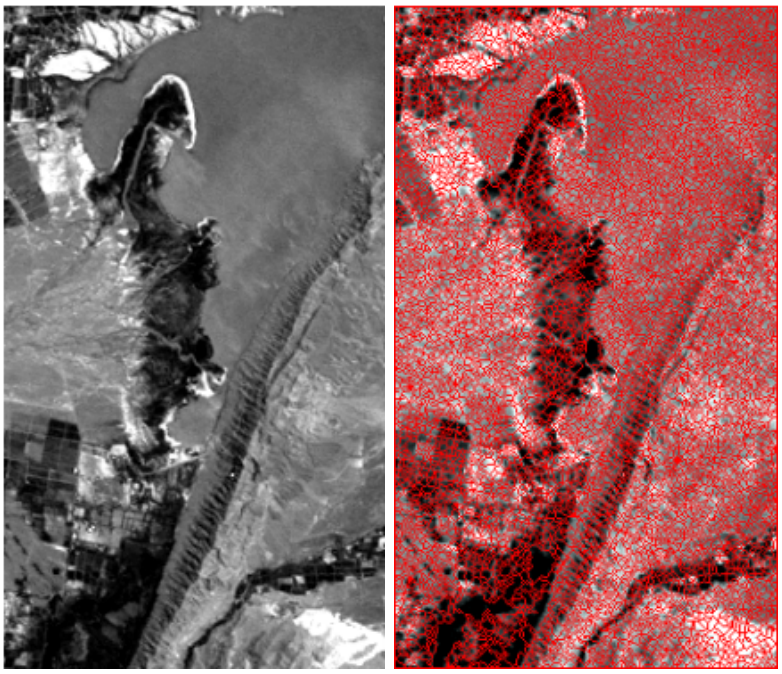


(a) (b)

Figura A.1: Watershed Lines. (a) Imagen original, (b) Se muestra el relieve topográfico, asumiendo que los puntos más oscuros son los que poseen nivel de gris inferior.

relieve. Ahora bien, en cada punto en que el agua proveniente de dos cuencas se mezcle, se construirán *represas*. Una vez que el relieve se encuentre completamente bajo el agua, quedarán las represas que dividen cada una de las regiones de la imagen. Las cuencas constituyen las Catchment Basins, mientras que las represas, las Watershed Lines. En la Figura A.2 se muestra la Transformación Watershed de una imagen.

Al observar los ejemplos, el lector puede notar claramente que se produce una sobresegmentación en la imagen. Se trata de un fenómeno muy conocido, y se debe a que cada mínimo, por más pequeño y angosto que sea, constituye su propia Catchment Basin, de aquí que el ruido y las pequeñas irregularidades presentes en la imagen produzcan este efecto tan notable. En este sentido, se limitará a mencionar que existen diversos métodos para apaliar este problema, pero la cobertura de dichos métodos está fuera del alcance de este trabajo.



(a) (b)

Figura A.2: Watershed Lines. (a) Imagen original, (b) Watershed lines superpuestas a la imagen original

.

Anexo B

Grafos

Un grafo es una estructura matemática, cuya utilidad es la representación de relaciones binarias entre elementos. Formalmente, es un par (E, Γ), donde E es un conjunto de elementos, llamados vértices o nodos, y Γ es un mapa de $E \to \rho(E)$, donde $\rho(E)$ es el conjunto de todos los subconjuntos de E, conocido en el campo como *partes de* E. Por ejemplo, si $E = \{1, 2, 3\}$, entonces $\rho(E) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$. Para cada $e \in E$, el conjunto de elementos $\Gamma(e)$ constituye el conjunto de vértices con los que e tiene relación, y se lo denomina *vecindario de* e.

Un grafo (E, Γ) se dice simétrico si, $\forall x, y \in E : x \in \Gamma(y) \implies y \in \Gamma(x)$. Si $x \in \Gamma(y)$ entonces se dice que el conjunto $\{x, y\}$ constituye un lado del grafo, y x es vecino de y.

Gráficamente, se representa un grafo con los vértices como puntos, y las lados como flechas que unen los puntos correspondientes, o segmentos si se trata de uno simétrico. En la Figura B.1 podemos observar la representación gráfica de un grafo simétrico (V, Γ), donde $V = \{A, B, C, D, E, F, G\}$ y $\Gamma(A) = \{B, D, G\}$.

Sea (E, Γ) un grafo, $X \subseteq E$ y $x_0, x_n \in X$, entonces

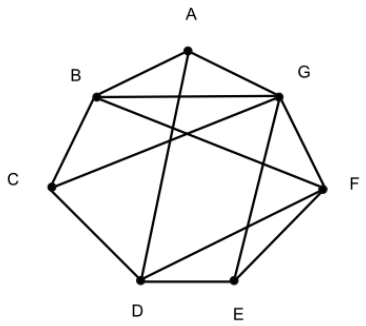


Figura B.1: Representación gráfica de un grafo.

- Un camino de x_0 hacia x_n en X será un conjunto ordenado $(x_0, x_1, ..., x_n)$ de elementos de X, donde $x_{i+1} \in \Gamma(x_i)$ para i = 0,1,...,n-1.
- Decimos que x_0, x_n están X-conectados si existe un camino de x_0 hacia x_n en X.
- Decimos que X es conectado si cada par de elementos de X están conectados en X.
- Sea $Y \subseteq E$, decimos que Y es una componente conectada de X si $Y \subseteq X$, Y está conectado y es maximal.
- (E, Γ) es conectado si E es conectado.

Podemos ejemplicar estas definiciones observando nuevamente el grafo (V, Γ) de la Figura B.1. Podemos ver que $\{A, B, C\}$ es un camino de A hacia C en V, y por lo tanto están V-conectados. Además, podemos decir que el conjunto $\{A, B, C\}$ es conectado, pero $\{A, B, C, E\}$ no lo es. Podemos mencionar que $\{A, B, C\}$ es una componente conectada de $\{A, B, C, E\}$. Finalmente, podemos decir que (V, Γ) es conectado.

Un *grafo ponderado* o *grafo con pesos* es un grafo en el que cada nodo tiene un peso asociado. Formalmente, es una estructura (E, Γ, F), donde (E, Γ) es un grafo y $F : E \to C$ es la función de peso, donde C usualmente es un conjunto numérico.

Bibliografía

[1] Luc Vincent and Pierre Soille *Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations*, 1991.

[2] Gilles Bertrand and Michel Couprie *Topological grayscale watershed transformation*, 1997.

[3] Laurent Najman and Michel Couprie *Watershed algorithms and contrast preservation*, 2003.

[4] Gilles Bertrand *On topological watersheds*, 2005.

[5] Michel Couprie, Laurent Najman and Gilles Bertrand *Quasi-linear algorithms for the topological watershed*, 2004.

[6] Michel Couprie and Laurent Najman *Quasi-linear algorithm for the component tree*, 2004.

[7] Michael A. Bender and Martín Farach-Colton *The LCA problem revisited*, 2000.

[8] Joel van Neerbos, Laurent Najman and Michael H.F. Wilkinson *Towards a Parallel Topological Watershed: First Results*, 2012.