

Índice general

1. Introducción	2
1.1. Motivación y Objetivos	2
1.2. Estado del Arte	2
1.3. Método de medición	2
1.4. Estructura del Trabajo	2
2. Transformación Watershed	3
2.1. Referencia histórica	3
2.2. Segmentación	3
2.3. Algoritmos	3
2.4. Aplicaciones	3
3. Watershed Topológico	4
3.1. Definiciones Preliminares	4
3.2. Definición	6
3.3. Bondades	6
3.4. Implementación Fuerza Bruta	6
3.5. Conclusión	8
4. Watershed Topológico Cuasi-Lineal	9
4.1. Definiciones preliminares	9
4.2. Árbol de Componentes	10
4.3. Algoritmo	12
4.4. Implementación	13
4.5. Conclusión	14
5. Watershed Topológico en Paralelo	15
5.1. Presentación del problema	15
5.2. Estado del arte	16
5.3. Algoritmo	17
5.4. Implementación	18
5.5. Conclusión	21
6. Conclusión	22

Capítulo 1

Introducción

1.1. Motivación y Objetivos

Pendiente

1.2. Estado del Arte

Pendiente

1.3. Método de medición

Pendiente

1.4. Estructura del Trabajo

Pendiente

Capítulo 2

Transformación Watershed

2.1. Referencia histórica

Pendiente

2.2. Segmentación

Pendiente

2.3. Algoritmos

Pendiente

2.4. Aplicaciones

Pendiente

Capítulo 3

Watershed Topológico

En este capítulo, se introducirá una variante de Transformación Watershed propuesta por M. Couprie y G. Bertrand [1], 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.

3.1. Definiciones Preliminares

Sea $k_{min}, k_{max} \in \mathbb{Z}$ donde $k_{min} < k_{max}$. Denotamos $\mathbb{K} = \{k \in \mathbb{Z} : k_{min} \leq k < k_{max}\}$. De aquí en adelante, (E, Γ) denotará un grafo, donde $E \subseteq \mathbb{Z}^2$. \mathcal{F} denotará el conjunto de todas las funciones de $E \rightarrow \mathbb{K}$. Si $F \in \mathcal{F}$ y $p \in E$ entonces definimos $\Gamma^-(p) = \{q \in \Gamma(p) : F(q) < F(p)\}$.

A continuación se presentan conceptos que el lector debe conocer para comprender los contenidos del presente capítulo. Si $F \in \mathcal{F}$, $k \in \mathbb{Z}$ entonces denominamos

- **k-sección superior**, o sección superior, al conjunto $F_k = \{x \in E; F(x) \geq k\}$.
- **k-sección inferior**, o sección inferior, al conjunto $\overline{F_k} = \{x \in E; F(x) < k\}$.
- **k-componente superior**, o componente superior, a una componente conectada de F_k .
- **k-componente inferior**, o componente inferior, a una componente conectada de $\overline{F_k}$.
- **mínimo regional**, o mínimo, de F a una k -componente inferior que no contenga una $(k-1)$ -componente inferior.

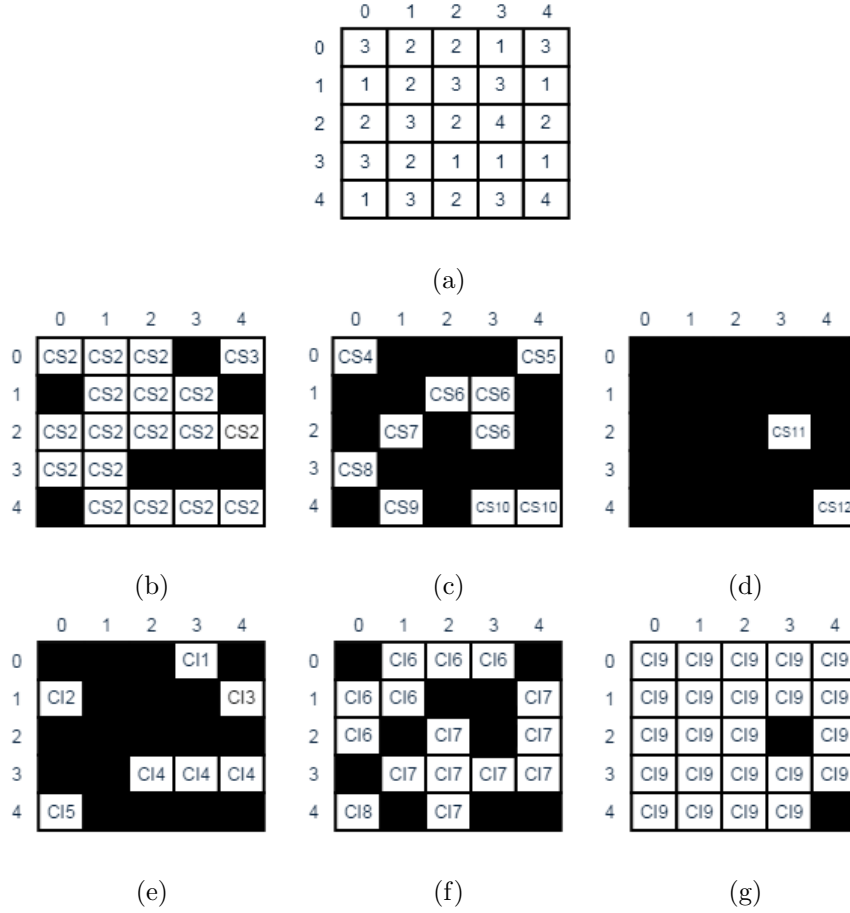


Figura 3.1: secciones superiores e inferiores. (a) Imagen original, (b) F_2 , (c) F_3 , (d) F_4 , (e) \overline{F}_2 , (f) \overline{F}_3 , (g) \overline{F}_4 . Las etiquetas reflejan las componentes que conforman la imagen

Podemos ilustrar estas definiciones observando la Figura 3.1, tomando una vecindad tipo 4.

Podemos observar que $k_{min} = 1, k_{max} = 5$. $F_1(\overline{F}_5)$ contendrá todos los puntos de la imagen, con lo que habrá una $1(5)$ -componente superior(inferior) compuesta por todos los puntos, a la que llamaremos CS1 (CI10). Además podemos observar que $F_5 = \overline{F}_1 = \emptyset$. A partir de las etiquetas de cada punto, podemos ver las componentes (superiores e inferiores) que constituyen la imagen, así, se observa que CI5 constituye una 2-componente inferior. Además, CI7, 3-componente inferior, no constituye un mínimo regional puesto que contiene a CI4, 2-componente inferior, que sí es un mínimo regional.

3.2. Definición

La definición formal de Watershed Topológico de una imagen, se basa en el concepto de punto W-simple. Sea V un conjunto no vacío y $X \subset V$ no vacío, entonces un punto $x \in X$ es **W-simple** para X si es adyacente a exactamente una componente conectada de \overline{X} .

Ahora bien, sea $F \in \mathcal{F}, p \in E, k = F(p)$, entonces, decimos que el punto p es **W-destructible** para F si es W-simple para F_k . Decimos que $G \in \mathcal{F}$ 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 de G y el resto de los puntos serán las watershed lines.

Continuando con el ejemplo de la Figura 3.1, podemos ver que el punto (4,2) es W-destructible, pues el valor del mismo es 2 y al observar $\overline{F_2}$, vemos que es adyacente sólo a CI4, mientras que el punto (1,3) no es W-destructible puesto que, como se observa en $\overline{F_3}$, es adyacente a CI6 y CI7.

3.3. Bondades

Si bien la transformación watershed de imágenes en escala de grises es una herramienta de gran utilidad y constituye un paso muy importante en la segmentación de imágenes, la mayoría de los enfoques existentes, tal como se menciona en [3], 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 $\overline{F_k}$. Más aún, se preserva los pass values para cada par de mínimos regionales de la imagen original. Esto constituye una gran virtud, y las implicaciones de la misma se encuentran detalladas en [2].

3.4. 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 imagen pixel por pixel, 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 pixel 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 pixel es W-destructible o no, asique comenzaremos por estudiar esta situación. 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 $q \in \Gamma^-(p)$ 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 calcular cada uno de los pixeles 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( $E, \Gamma, F, p$ )
2:   if  $\Gamma^-(p) = \emptyset$  then return false
3:   if  $|\Gamma^-(p)| = 1$  then return true
4:    $reachableNodes \leftarrow \emptyset$ 
5:    $index \leftarrow 0$ 
6:    $reachableNodes \cdot insert(\Gamma^-(p)[0])$ 
7:   while  $\neg(reachableNodes \cdot containAll(\Gamma^-(p))) \wedge index <$ 
      $reachableNodes \cdot size$  do
8:      $q \leftarrow reachableNodes[index]$ 
9:     for all  $n \in \Gamma(q)$  do
10:      if  $F(n) < F(p)$  then
11:         $reachableNodes \cdot insert(n)$ 
12:       $index \leftarrow index + 1$ 
   return  $reachableNodes \cdot containAll(\Gamma^-(p))$ 

```

Se recibe por parámetro un grafo 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 irá inser-

tando 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 $\Gamma^-(p) \subseteq \text{reachableNodes}$, esto significa que todos los $q \in \Gamma^-(p)$ pertenecen 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 lo 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-destructibles, 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.

Algorithm 2

```

1: procedure DO_TOPOLOGICAL_WATERSHED( $E, \Gamma, F$ )
2:   repeat
3:      $\text{someChange} \leftarrow \text{false}$ 
4:     for all  $p \in E$  do
5:       if  $\text{isWdestructible}(E, \Gamma, F, p)$  then
6:          $F(p) \leftarrow F(p) - 1$ 
7:          $\text{someChange} \leftarrow \text{true}$ 
8:   until  $\text{someChange} = \text{true}$ 

```

3.5. Conclusión

Pendiente

Capítulo 4

Watershed Topológico Cuasi-Lineal

En [4] se propone un algoritmo de orden cuasi-lineal para el cálculo del Watershed Topológico 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 último 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.

4.1. Definiciones preliminares

A continuación se presentan conceptos clave que se utilizarán 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 3.1, podemos ver que $C_2(F)$ está compuesto por CS2 y CS3, mientras que $C(F)$ está

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.

4.2. Árbol de Componentes

Imaginemos una imagen, como un relieve topográfico completamente sumergido en agua al que se le hacen perforaciones en sus mínimos. El nivel de agua irá disminuyendo lentamente, con lo que comenzaremos 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 continúe descendiendo, las islas irán creciendo 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.

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** 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 $\Psi : E \rightarrow C(F)$, que relaciona cada $p \in E$ con el elemento de $C(F)$ que lo contiene.

Algo muy importante a cerca 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 4.1 vemos el mapeo de componentes junto al Árbol de componentes correspondiente.

A continuación se presentan la definición de "highest fork" del árbol. 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 .

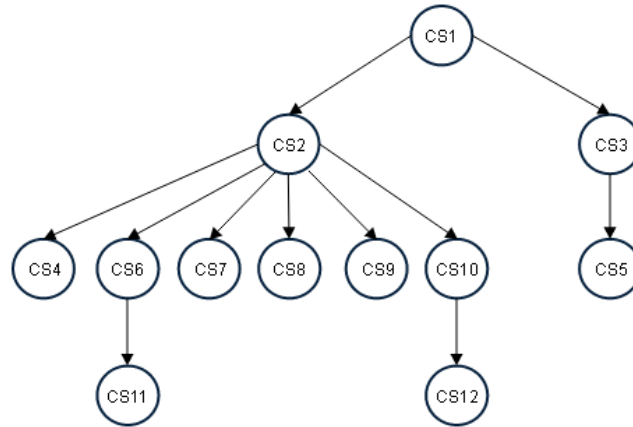
Sea $F \in \mathcal{F}$ y $[k, c], [k_1, c_1], [k_2, c_2], \dots, [k_n, c_n] \in C(F)$, entonces

- $[k_1, c_1]$ es **ancestro** de $[k_2, c_2]$ si $k_2 \leq k_1$ y $c_2 \subseteq c_1$. 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 común** 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 $[k, c]$.

¹Para más información consultar [5]

	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 4.1: (a) Mapeo de Componentes, (b) Árbol de Componentes. En ambos casos respecto a la imagen de la Figura 3.1 (a)

- $[k, c]$ 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 $M = \{[k_1, c_1], [k_2, c_2], \dots, [k_n, c_n]\}$ 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 $[k_i, c_i], [k_j, c_j]$.

Podemos tomar como ejemplo el Árbol de la Figura 4.1, podemos 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 tomamos $\{CS2, CS3\}$ su highest fork es CS1.

4.3. Algoritmo

En este apartado vamos a explicar cómo utilizar el Árbol de Componentes en el cálculo de la transformación.

Sea $F \in \mathcal{F}$, para calcular el Watershed Topológico de F vamos a calcular $T(\bar{F})$ teniendo en cuenta que $\bar{F}(p) = F(p) + 1$ ². En este caso habrá una arista de $c' \in C_k(\bar{F})$ hacia la componente $c \in C_j(\bar{F})$ 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 [4].

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

- Si $V(p) \neq \emptyset$ y $V(p)$ no tiene highest fork en $T(\bar{F})$, 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 $T(\bar{F})$ 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 $w - 1$.

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

Este resultado es de gran importancia, pues permite reducir las iteraciones del algoritmo fuerza bruta enormemente. 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.

4.4. 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 disminuimos los valores de cada uno de los puntos W-destructibles.

En base a esta observación, vamos a definir 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.

Algorithm 3

```

1: function ISWDESTRUCTIBLE( $E, F, \Gamma, p, T(\overline{F}), \Psi$ )
2:    $V \leftarrow \emptyset$ 
3:   for all  $q \in \Gamma^-(p)$  do
4:      $V \cdot insert(\Psi(q))$ 
5:   if  $V = \emptyset$  then return null
6:    $[k, c] \leftarrow highestFork(T(\overline{F}), V)$ 
7:   if  $[k, c] = null$  then return minLevel( $V$ )
8:   if  $k \leq F(p)$  then return  $[k, c]$ 
   return null

```

Cabe aclarar, que en [4] 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 [6].

Ahora pasamos a detallar el algoritmo que lleva a cabo la transformación. Vamos a utilizar en nuestro algoritmo un mapa $: E \rightarrow C(\overline{F})$, que asocia cada p con el nodo retornado por la función `isWdestructible`. Para procesar cada

uno de los $p \in E$ en el orden conveniente, lo único que vamos a necesitar que ese mapa este ordenado de modo creciente con respecto al nivel del nodo.

Pasamos ahora a explicar 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 e ir extrayendo 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 la situación 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 descrito.

Algorithm 4

```

1: procedure DO_TOPOLOGICAL_WATERSHED( $E, \Gamma, F, T(\overline{F}), \Psi$ )
2:    $componentMap \leftarrow \emptyset$ 
3:   for all  $p \in E$  do
4:      $[i, c] \leftarrow isWdestrucible(E, F, \Gamma, p, T(\overline{F}), \Psi)$ 
5:     if  $\neg[i, c] = null$  then
6:        $componentMap \cdot insert(p, [i, c])$  ▷ sorted by i
7:   while  $\neg componentMap = \emptyset$  do
8:      $p, [i, c] \leftarrow componentMap \cdot extractFirst()$ 
9:      $F(p) \leftarrow i - 1$ 
10:     $\Psi(p) \leftarrow [i, c]$ 
11:    for all  $q \in \Gamma(p), F(q) \geq i$  do ▷ if  $F(q) < i$ , then  $q$  has already
      been processed so can not be W-destructible
12:       $[j, d] \leftarrow isWdestrucible(E, F, \Gamma, q, T(\overline{F}), \Psi)$ 
13:      if  $[j, d] = null$  then
14:         $componentMap \cdot removeIfExists(q)$ 
15:      else
16:         $componentMap \cdot addOrUpdate(q, [j, d])$ 

```

4.5. Conclusión

Pendiente

Capítulo 5

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 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.

5.1. Presentación del problema

Básicamente, el algoritmo cuasi-lineal descrito 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, scanear 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 propondremos en este trabajo, se basa en dividir en fragmentos la imagen a transformar y correr, 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 en este trabajo se dividirá la imagen de modo de distribuir, en cada fragmento, los puntos del modo más equitativo posible en términos de cantidad y de manera de minimizar la cantidad de puntos adyacentes a fragmentos ajenos. Un ejemplo de esta división se muestra en la Figura 5.1.

El primer paso mencionado previamente, puede hacerse sin problemas puesto que la imagen sólo se lee y, por lo tanto, no hay problemas de sin-

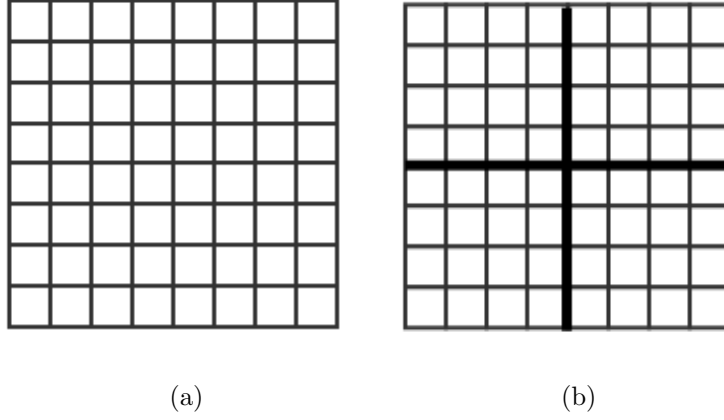


Figura 5.1: (a) Imagen original. (b) Una posible división en 4 fragmentos.

cronización. Respecto del segundo, supongamos que tenemos 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 -destruible 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 podemos estar seguros del valor al que debe ser disminuido p . Del mismo modo, si disminuyéramos p , quedarán desactualizados los mapas de los fragmentos en los que p tenga vecinos. Debido a estos problemas, podemos terminar con una imagen resultante que no satisface la definición de watershed topológico.

De aquí en adelante, diremos 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 5.2 se observan los puntos bordes de la Figura 5.1 tomando una relación de vecindad tipo 4 o tipo 8.

5.2. Estado del arte

Como vimos 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 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

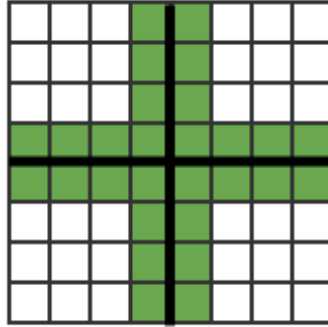


Figura 5.2: En verde se muestran los puntos borde de la Figura 5.1, tomando una relación de vecindad tipo 4 o tipo 8.

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 [7] 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 $i+1$, y así hasta procesar todos los subfragmentos.

Claro, 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 algún otro subfragmento se vuelva W-destructible. Por este motivo, se utiliza un array global para marcar los puntos que hayan sido modificados, y así 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 5.3.

5.3. 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 descrito 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.

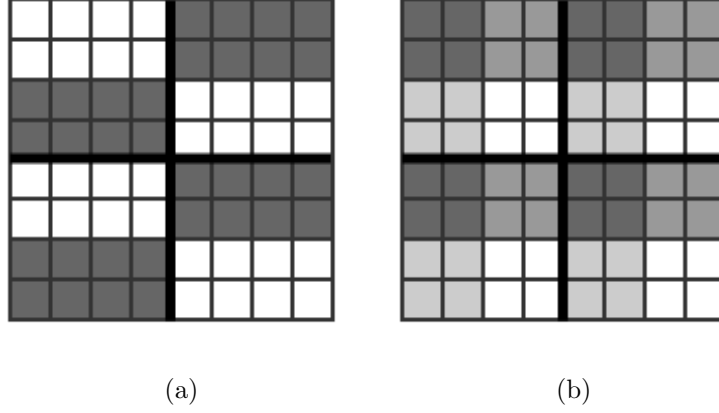


Figura 5.3: Subfragmentación para Figura 5.1. (a) Subfragmentación para el caso de una relación de vecindad de tipo 4. Cada fragmento se divide en 2 subfragmentos. (b) Subfragmentación para el caso de una relación de vecindad de tipo 8. Cada fragmento se divide en 4 subfragmentos.

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.

5.4. Implementación

A continuación presentaremos 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( $E, \Gamma, F, T(\overline{F}), \Psi, Tile$ )
2:   for all  $p \in Tile$  do
3:      $[i, c] \leftarrow isWdestructible(E, F, \Gamma, p, T(\overline{F}), \Psi)$ 
4:     if  $\neg[i, c] = null$  then
5:        $componentMap \cdot insert(p, [i, c])$  ▷ 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 a 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 DO_TOPOLOGICAL_WATERSHED_ON_TILE( $E, \Gamma, F, T(\overline{F}), \Psi, Tile$ )
2:    $componentMap \leftarrow InitializeQueue(E, \Gamma, F, T(\overline{F}), \Psi, Tile)$ 
3:   while  $\neg componentMap = \emptyset$  do
4:      $p, [i, c] \leftarrow componentMap \cdot extractFirst()$ 
5:     if  $\neg \Gamma(p) \subseteq Tile$  then ▷ means a border px
6:        $borders \cdot insert(p)$ 
7:     else
8:        $F(p) \leftarrow i - 1$ 
9:        $\Psi(p) \leftarrow [i, c]$ 
10:      for all  $q \in \Gamma(p), F(q) \geq i$  do
11:         $[j, d] \leftarrow isWdestructible(E, F, \Gamma, q, T(\overline{F}), \Psi)$ 
12:        if  $[j, d] = null$  then
13:           $componentMap \cdot removeIfExists(q)$ 
14:        else
15:           $componentMap \cdot addOrUpdate(q, [j, d])$ 

```

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 DO_TOPOLOGICAL_WATERSHED_ON_BORDER( $E, \Gamma, F, T(\overline{F}), \Psi$ )
2:    $componentMap \leftarrow InitializeQueue(E, \Gamma, F, T(\overline{F}), \Psi, borders)$ 
3:   while  $\neg componentMap = \emptyset$  do
4:      $p, [i, c] \leftarrow componentMap \cdot extractFirst()$ 
5:      $F(p) \leftarrow i - 1$ 
6:      $\Psi(p) \leftarrow [i, c]$ 
7:     for all  $q \in \Gamma(p), F(q) \geq i$  do
8:        $[j, d] \leftarrow isWdestructible(E, F, \Gamma, q, T(\overline{F}), \Psi)$ 
9:       if  $[j, d] = null$  then
10:         $componentMap \cdot removeIfExists(q)$ 
11:         $changed \cdot removeIfExists(q)$ 
12:       else
13:        if  $q \in componentMap$  then
14:           $componentMap \cdot update(q, [j, d])$ 
15:        else
16:           $changed \cdot insert(q)$ 

```

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 temrnar 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 global *changed*.

Finalmente, aquí se presenta el procedimiento encargado de hacer la transformación watershed paralelo.

Algorithm 8

```

1: procedure DO_TOPOLOGICAL_WATERSHED( $E, \Gamma, F, T(\overline{F}), \Psi, n$ )
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:      $threadPool[i].run(doTopologicalWatershedOnTile(E, \Gamma, F, T(\overline{F}), \Psi, tiles[i]))$ 
8:    $waitForAllThreadsToFinish()$ 
9:    $doTopologicalWatershedOnBorder(E, \Gamma, F, T(\overline{F}), \Psi)$ 
10:  while  $\neg changed = \emptyset$  do
11:    for  $i = 1 \rightarrow n$  do
12:       $threadPool[i].run(doTopologicalWatershedOnTile(E, \Gamma, F, T(\overline{F}), \Psi, tiles[i] \cap$ 
         $changed))$ 
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 del 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, y así hasta lograr estabilidad.

5.5. Conclusión

Pendiente

Capítulo 6

Conclusión

Pendiente

Bibliografía

- [1] Gilles Bertrand and Michel Couprie *Topological grayscale watershed transformation*, 1997.
- [2] Laurent Najman and Michel Couprie *Watershed algorithms and contrast preservation*, 2003.
- [3] Gilles Bertrand *On topological watersheds*, 2005.
- [4] Michel Couprie, Laurent Najman and Gilles Bertrand *Quasi-linear algorithms for the topological watershed*, 2004.
- [5] Michel Couprie and Laurent Najman *Quasi-linear algorithm for the component tree*, 2004.
- [6] Michael A. Bender and Martín Farach-Colton *The LCA problem revisited*, 2000.
- [7] Joel van Neerbos, Laurent Najman and Michael H.F. Wilkinson *Towards a Parallel Topological Watershed: First Results*, 2012.