
Paralelización mediante OpenMP de segmentación de imágenes para el análisis de materiales bidimensionales

Grado en Ingeniería Informática

Ingeniería de Computadores

Proyecto de Fin de Grado

2015

Autor:

Daniel Franco Barranco

Supervisores:

Ibai Gurrutxaga Goikoetxea

Javier Muguerza Rivero



Universidad
del País Vasco

Euskal Herriko
Unibertsitatea



informatika
fakultatea

facultad de
informática

Agradecimientos

Estos son los agradecimientos.

Índice general

1. Introducción	1
1.1. Motivación	1
1.2. Estructura de la memoria	2
1.3. Objetivos y necesidades del cliente	3
2. Conceptos básicos	7
2.1. Definición de segmentación	7
2.2. Evolución de la segmentación	7
2.3. Aplicaciones	9
3. Técnicas de segmentación	11
3.1. Introducción	11
3.2. Segmentación de imagen basada en el tratamiento de los <i>píxeles</i>	12
3.2.1. Thresholding	12
3.2.2. Clustering	15
3.2.3. Morphology	16
3.3. Segmentación de imagen basada en la detección de bordes . . .	17
3.3.1. Detección de borde usando gradientes	18
3.3.2. Active Contours	20
3.3.3. <i>Level set</i>	22
3.4. Segmentación de imagen basada en regiones crecientes	24
3.4.1. <i>Region growing</i>	24
3.4.2. <i>Split/Merge</i>	29
3.4.3. <i>Graph cut</i>	29
3.5. Conclusiones de los tipos de segmentación	30
4. Optimizaciones y mejoras del algoritmo <i>level set</i> original	31
4.1. Introducción	31
4.2. Aproximación a la técnica de <i>level set</i>	32
4.3. Evolución del contorno	38

5. Implementación de la aproximación y Ofeli	39
5.1. Ofeli	39
5.1.1. Estructura	39
5.1.2. Esquema general	41
6. Paralelización de la aproximación del algoritmo <i>level set</i>	43
6.1. Planteamiento de la paralelización	44
6.1.1. Primera implementación	47
6.1.2. Rendimiento	51
6.2. Primer paso de mejora: segunda implementación	53
6.2.1. Rendimiento	54
6.3. Segundo paso de mejora: tercera implementación	55
6.3.1. Rendimiento	57
7. Experimentación	61
7.1. Primera imagen	62
7.2. Segunda imagen	66
7.3. Tercera imagen	70
8. Gestión del proyecto	75
8.1. Gestión del alcance	75
8.2. Gestión del tiempo	75
8.3. Gestión de los riesgos	75
8.4. Gestión del costes	75
9. Conclusiones	77
9.1. Líneas futuras	77
Bibliografía	79

Índice de figuras

1.1. Ejemplo de proyecciones en láminas de materiales sintetizados mediante la deposición química de vapor	4
2.1. Resultados de búsqueda en el sitio web <i>ieee explore</i> [1] con las palabras «image segmentation»	8
3.1. Clasificación de las técnicas de segmentación	12
3.2. Histograma que muestra tres aparentes segmentos de la imagen con dos umbrales	13
3.3. Segmentación con <i>thresholding</i>	14
3.4. Segmentación con <i>thresholding</i> con varios umbrales	14
3.5. Segmentación con <i>thresholding</i> en imagen con iluminación gradiante	14
3.6. Segmentación utilizando K-means con K=16	16
3.7. Segmentación utilizando K-means con K=6	16
3.8. Ejemplo de dilatación de la técnica de morfología	17
3.9. Detección de bordes con el uso de los operadores	20
3.10. Ejemplo de un contorno activo	21
3.11. Ejemplo de segmentación de plantas presentando en [2] utilizando los contornos activos	22
3.12. Ilustración de la superficie de <i>level set</i>	23
3.13. Ilustración del método de <i>level set</i>	24
3.14. Ejemplo del funcionamiento de la técnica de <i>region growing</i> . .	25
3.15. Ejemplo de la técnica de <i>region growing</i>	26
3.16. Ejemplo del funcionamiento de la técnica de <i>watershed</i>	28
3.17. Funcionamiento de la técnica de <i>watershed</i>	28
3.18. Funcionamiento de la técnica de <i>watershed</i>	28
3.19. Funcionamiento de la técnica de <i>graph cut</i> en una imagen 3x3	30
4.1. Representación de la curva C	33

4.2. Ejemplo de expansión del contorno un píxel hacia fuera. Operaciones que se realizan continuamente en el paso 2 y 3 del algoritmo	37
4.3. Posibles inicializaciones del contorno con el método de Chan-Vese	38
5.1. Esquema de las funciones utilizadas en el trabajo de Ofeli para realizar el algoritmo <i>level set</i> aproximado	41
6.1. Ejemplo de condición de carrera al querer añadir un punto vecino en común a puntos tratados por distintos <i>threads</i>	46
6.2. Esquema de la primera implementación	47
6.3. Esquema de la colocación de los <i>heads</i> de trozo de lista a crear	48
6.4. Ejemplo de creación de cada sublistas usando los <i>heads</i> establecidos	48
6.5. Ejemplo de unión de cada sublistas usando los <i>heads</i> y <i>tails</i> de cada una	49
6.6. Ejemplo de unión de cada sublistas usando los <i>heads</i> y <i>tails</i> de cada una	49
6.7. Gráfica de los tiempos de ejecución de la primera implementación paralela	52
6.8. Gráfica de los tiempos de ejecución de la primera y segunda implementación	54
6.9. Ejemplo de una malla de cuatro locks de establecidos en forma rectangular	55
6.10. Gráfico de los resultados obtenidos en este tercera implementación	57
6.11. Gráfica de los tiempos de ejecución con los <i>locks</i> y la estructura de la «sección crítica normal»	58
7.1. Tipos de imágenes con las que se realizarán las pruebas completas de la tercera implementación	62
7.2. Gráfica de los tiempos de ejecución de la imagen 7.1a con un tamaño de 500x500	63
7.3. Gráfica de los tiempos de ejecución de la imagen 7.1a con un tamaño de 1500x1500	64
7.4. Gráfica de los tiempos de ejecución de la imagen 7.1a con un tamaño de 3000x3000	65
7.5. Resultado de la segmentación de la imagen 7.1a	66
7.6. Gráfica de los tiempos de ejecución de la imagen 7.1b con un tamaño de 500x500	67

ÍNDICE DE FIGURAS

VII

7.7. Gráfica de los tiempos de ejecución de la imagen 7.1b con un tamaño de 1500x1500	68
7.8. Gráfica de los tiempos de ejecución de la imagen 7.1b con un tamaño de 3000x3000	69
7.9. Resultado de la segmentación de la imagen 7.1b	70
7.10. Gráfica de los tiempos de ejecución de la imagen 7.1c con un tamaño de 500x500	71
7.11. Gráfica de los tiempos de ejecución de la imagen 7.1c con un tamaño de 1500x1500	72
7.12. Gráfica de los tiempos de ejecución de la imagen 7.1c con un tamaño de 3000x3000	73
7.13. Resultado de la segmentación de la imagen 7.1c	74

VIII

ÍNDICE DE FIGURAS

Índice de tablas

3.1. Máscaras utilizadas por el operador de Roberts de tamaño 2x2	19
3.2. Máscaras utilizadas por el operador de Sobel de tamaño 3x3 .	19
3.3. Máscaras utilizadas por el operador de Prewitt de tamaño 3x3	19
4.1. Algoritmo completo de la aproximación del <i>level set</i>	36
6.1. Rendimiento de las ejecuciones de la primera implementación paralela	51
6.2. Desglose de tiempos (s) de un ciclo del algoritmo con una ejecución paralela con dos <i>cores</i> de la primera implementación	52
6.3. Rendimiento obtenido de las ejecuciones de la segunda implementación paralela	54
6.4. Tiempos de ejecución (s), eficiencia y <i>speed-ups</i> (su) obtenidos con distintas combinaciones de <i>threads</i> y <i>locks</i> en la tercera implementación paralela	57
7.1. Tiempos de ejecución de la imagen 7.1a con un tamaño de 500x500	63
7.2. Tiempos de ejecución de la imagen 7.1a con un tamaño de 1500x1500	64
7.3. Tiempos de ejecución de la imagen 7.1a con un tamaño de 3000x3000	65
7.4. Tiempos de ejecución de la imagen 7.1b con un tamaño de 500x500	67
7.5. Tiempos de ejecución de la imagen 7.1b con un tamaño de 1500x1500	68
7.6. Tiempos de ejecución de la imagen 7.1b con un tamaño de 3000x3000	69
7.7. Tiempos de ejecución de la imagen 7.1c con un tamaño de 500x500	71

7.8. Tiempos de ejecución de la imagen 7.1c con un tamaño de 1500x1500	72
7.9. Tiempos de ejecución de la imagen 7.1c con un tamaño de 3000x3000	73

Resumen

Puedes poner un resumen de lo que trata el PFC.

Abstract

Puedes poner un resumen de lo que trata el PFC.

Laburpena

Puedes poner un resumen de lo que trata el PFC.

Capítulo 1

Introducción

Este proyecto se ha desarrollado para dar respuesta a un proyecto emergente dentro de la fundación Donostia International Physics Center (DIPC) llamado Morfokinetics. Por ello, ciertas de las actividades a desarrollar han sido propuestas por dicho proyecto ya que son necesarias para su avance y desarrollo. Así pues, se ha mantenido contacto con el director del proyecto interno del DIPC durante el ciclo de vida de este proyecto.

El objetivo de este proyecto es el análisis computacional y experimental de la segmentación de imágenes, en este caso, producidas mediante la deposición química de vapor de materiales sintetizados. En el tratamiento de estas imágenes, aparte del objetivo principal de poder encontrar los diferentes elementos que se encuentran en ella, también se presentan varias tareas complementarias: porcentaje de recubrimiento de los elementos encontrados, aislamiento y densidad espectral de cada elemento.

Una vez realizadas estas tareas, se requiere parallelizar el tipo de técnica de segmentación escogida para poder minimizar el tiempo de espera al resultado por parte del usuario (cliente) o, incluso, poder tratar los *frames* de una secuencia de vídeo de manera que se pueda dar una respuesta en tiempo real.

1.1. Motivación

Como se explicará más adelante en el apartado 2.2 la segmentación de imágenes es algo que ha avanzado mucho durante las últimas décadas. Esto se debe a que muchas de las tecnologías que utilizamos diariamente hacen uso en cierta medida de la segmentación de imagen para realizar ciertas funciones. La segmentación de imagen está presente en la detección de movimiento de objetos, en sistemas de posicionamiento humano, en las consolas, en aplica-

ciones fotográficas en las que se detectan los rostros humanos o se reconocen objetos, en imágenes de satélites espaciales para la predicción meteorológica, en la localización de tumores u otros síntomas patológicos en radiografías médicas etc.

A parte de ayudar a mejorar ciertas funciones de la vida cotidiana, la segmentación se ha vuelto en muchos campos, como la medicina, indispensable para hacer ciertos trabajos. Por no decir, que es la pieza clave en trabajos como la cirugía robotizada, ya que es necesario que, al estar realizando la operación, se detecten con precisión todas las partes físicas del cuerpo humano para no poner en riesgo la vida del paciente. Otra utilidad también sería la reconstrucción craneal y/o cerebral en 3D de pacientes.

Las técnicas de segmentación más avanzadas consiguen información detallada sobre la imagen tratada logrando así un resultado mejor, sin embargo cuanta más precisión se necesita más costoso es el algoritmo. Por lo tanto, muchas veces se suele buscar un equilibrio entre la precisión necesaria y la respuesta temporal mínima exigida a la hora de elegir un algoritmo adecuado. Por esta razón, hay muchos trabajos que tratan de mejorar esta respuesta temporal de los algoritmos de segmentación en base a optimizaciones. El resultado son miles de artículos relacionados con implementaciones paralelas e implementaciones en GPU (*Graphics Processing Unit*) aparte de optimizaciones matemáticas y algorítmicas.

En conclusión, la paralelización de métodos de segmentación está a la «orden del día» puesto que es necesaria una respuesta rápida en determinados ámbitos de uso. Este trabajo pretende crear una versión paralela robusta sobre una aproximación al método de *level set* para proporcionar una respuesta rápida sobre este tipo de segmentación.

1.2. Estructura de la memoria

Este documento está estructurado en varios apartados.

La primera parte de la documentación es el entorno de investigación del proyecto que está formada por el primer, segundo, tercer y cuarto capítulo. Se presentan varios conceptos básicos: la definición de la segmentación y la evolución histórica de ésta. Realizada esta introducción, se comentan las aplicaciones que hoy en día tiene este ámbito que, como se podrá observar, serán muchas. Con esta primera parte se espera poder ubicar al lector sobre este análisis de imágenes, pieza angular de este proyecto, para que pueda entender correctamente el resto de la documentación. Más adelante se presenta una clasificación de los diferentes tipos de segmentación de imágenes, explicando individualmente sus características y algunos trabajos que se hayan

realizado de estas técnicas. El final de esta primera parte se centra en una técnica de segmentación concreta: *level set*. Se comentarán variaciones de la técnica y mejoras hasta llegar a una aproximación la cuál mejorará el tiempo de ejecución significativamente y con la que se ha trabajado en el proyecto. El desarrollo realizado en este proyecto está basado en el trabajo publicado en [3].

En la segunda parte del documento, formada por el quinto, sexto y séptimo capítulo, se encuentra todo lo relacionado con la paralelización de la técnica utilizada. La primera toma de contacto o la primera propuesta de paralelización creada, las posteriores fases desarrolladas, las conclusiones y decisiones tomadas en cada fase y el rendimiento obtenido en la versión final.

En la tercera y última parte, formada por el octavo y noveno capítulo, se presentan las conclusiones del trabajo realizado, algunos aspectos de la gestión del mismo y varias propuesta de mejora. Al final del documento puede encontrarse la bibliografía y los anexos.

1.3. Objetivos y necesidades del cliente

La primera toma de contacto con el director del proyecto Morfokinetics de la fundación del DIPC fue en el mes de febrero. Parte de ese proyecto lo realizó la compañera Estíbaliz Sánchez, por lo que se encontrará más información general de él en la memoria de su PFG [4]. A lo largo de la documentación de este PFG el director del proyecto Morfokinetics aparecerá nombrado como el cliente, ya que es la persona a la que se le dará respuesta cumpliendo los objetivos de este proyecto.

En el proyecto Morfokinetics hay una parte en la que es necesario realizar un tratamiento de una imagen para sacar de ésta varias conclusiones. El cliente conoce la teoría sobre un algoritmo llamado *level set* que realiza segmentación de imágenes con bastante precisión y que está bastante extendido. Sin embargo, no conoce ninguna implementación de este algoritmo ni el tiempo que puede costar realizar dicha segmentación. Recordando la introducción del proyecto, las imágenes se han sacado de una proyección en cierta lámina de materiales sintetizados mediante la deposición química de vapor, por lo que tendrán una apariencia similar a las presentadas en la figura 1.1. La principal tarea u objetivo que se deberá realizar en este proyecto para satisfacer las necesidades del cliente es la siguiente:

1. Realizar la segmentación lo más rápidamente posible. De esta manera la técnica de segmentación podrá ser capaz de analizar un vídeo de la evolución de la deposición de estos materiales *frame a frame*.

A parte del principal objetivo, el cliente también necesita varias tareas alternativas una vez realizada la segmentación de la imagen:

1. Determinar el número de islas existentes
2. Porcentaje de recubrimiento de las islas respecto al tamaño de la imagen
3. Aislamiento de cada isla
4. Densidad espectral de cada isla
5. Por todo ello, habrá que encontrar las islas¹ contenidas en la imagen

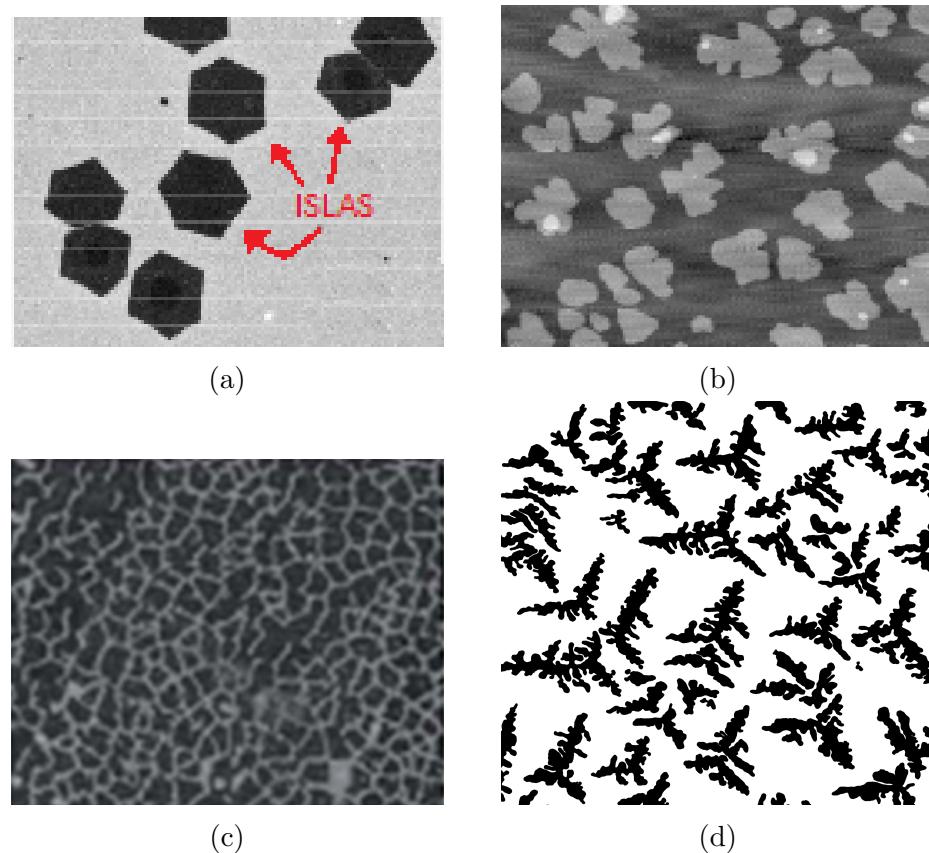


Figura 1.1: Ejemplo de proyecciones en láminas de materiales sintetizados mediante la deposición química de vapor

¹se llamará «isla» a cada superficie que se distinga sobre el fondo de la imagen. Ver ejemplo en 1.1a

Las islas puede que estén muy separadas, como se puede ver en la imagen 1.1a o muy juntas como en la imagen 1.1c. También se puede observar que la forma de estas islas no es siempre la misma. Estas características dependen de ciertas variables físicas que hacen que el material se «pegue» de una determinada manera en la lámina, dando lugar a estas imágenes abstractas.

Capítulo 2

Conceptos básicos

2.1. Definición de segmentación

La segmentación de imagen, también denominada a veces como *labelling*, es el proceso de dividir la imagen en grupos o regiones contiguas cuyos elementos(p. e. píxeles o *voxels*) tienen propiedades o características comunes. Estas regiones servirán para identificar los objetos de la imagen que posteriormente podrán ser clasificados y etiquetados en base a sus propiedades [5].

El resultado final de la segmentación de la imagen será un conjunto de regiones o segmentos que formarán la imagen original. Cada uno de los píxeles de una región tendrá una característica común con los píxeles de dicha región y una diferencia significativa respecto a píxeles de otra región, por ello se habrán agrupado en distinto segmento, ya sea por ejemplo, en el color, la textura o la intensidad. Por lo tanto, se podrán extraer los segmentos de interés de la imagen, es decir, los objetos que ésta contiene.

2.2. Evolución de la segmentación

Los primeros desarrollos en el ámbito de la segmentación de imagen se remontan a hace 50 años. En 1965 se desarrolló un operador para detectar bordes entre diferentes partes de una imagen, conocido como *Roberts operator* o *Roberts Edge Detector*. Este detector fue el primer paso hacia la descomposición de imágenes en diferentes segmentos o regiones. En esa misma década también se propusieron varios detectores de bordes como *Sobel* y *Prewitt edge detectors*. A partir de ahí, comenzaron a surgir diferentes algoritmos y técnicas de segmentación. Junto con esto, también se amplió el ámbito de estas técnicas: de imagen 2D a 3D, de imágenes fijas a imágenes en «movimiento»

o secuencias de imágenes, de escalas de gris a imágenes a color etc [6].

A pesar de los años de investigación dedicados a estas técnicas y el gran número de ellas existentes, la segmentación de imagen sigue siendo un tema de investigación desafiante y no existe aún un estándar de segmentación que funcione bien para cualquier tipo de imagen. Estas técnicas están en continua evolución y aún están lejos de su madurez. Prueba de ello está en que muchas conferencias del ámbito de tratamiento de imagen tienen apartados de segmentación, además, el número de artículos de este ámbito aumenta cada año y muchos libros de procesamiento de imagen tienen capítulos referidos a la segmentación [6].

Para que el lector se haga una idea, se ha realizado una búsqueda en el sitio web *ieee explore*[1] con las palabras «image segmentation» en varios años. Este buscador encuentra artículos, conferencias, estándares, libros, revistas y cursos de aprendizaje relacionados con las palabras introducidas. La figura 2.1 muestra el número de resultados de esa búsqueda desde 1960 hasta 2015. Como se puede observar, el número de resultados aumenta notablemente cada lustro.



Figura 2.1: Resultados de búsqueda en el sitio web *ieee explore* [1] con las palabras «image segmentation»

Fuente: [1]

2.3. Aplicaciones

Las aplicaciones de segmentación de imágenes son muchas y muy diversas. Cualquier proceso que requiera la extracción de información de una imagen utilizará, en cierta medida, una técnica de segmentación. A continuación nombraremos algunas de las aplicaciones que se han ido recopilando en la realización de esta documentación, aunque el número de aplicaciones totales es mucho mayor:

- Localización de moléculas en imágenes microscópicas.
- Aplicaciones médicas.
 - Localización de tumores y otras patologías.
- Detección de cuerpos para aplicaciones de seguimiento de movimientos como Kinect.
 - Operaciones guiadas por ordenador.
- Localización de objetos en imágenes de satélite.
- Visión por computador.
- Reconocimientos faciales.
- Reconocimiento de plantas

Capítulo 3

Técnicas de segmentación

3.1. Introducción

Hay bastante controversia en cuanto a la clasificación de las diferentes técnicas de segmentación, por el gran número de estas técnicas existentes, por las diferentes maneras en las que cada una tiene representada la imagen, las diferentes características que utilizan de la imagen etc. Hay trabajos que realizan una clasificación de los algoritmos desde dos puntos de vista diferentes: en función de cómo puede ser utilizado el algoritmo, es decir, las aplicaciones que pueda tener, y otra en base al algoritmo en sí, fijándose en cómo realiza la segmentación [7]. Por otro lado, también hay otros trabajos que realizan esta clasificación para ámbitos muy concretos [7, pag. 11].

Zhang[6] propone una clasificación más clara, creando la división en dos grupos: a) algoritmos basados en detectar la discontinuidad de las diferentes regiones de la imagen, los llamados *edge-based algorithms* b) basados en detectar la continuidad o la similitud de las regiones, los llamados *region-based algorithms*. Posteriormente hace una subdivisión de estos dos grupos en función de la estrategia de procesamiento: los que realizan un procesamiento secuencial, donde el procesamiento de pasos previos se tienen en cuenta en pasos posteriores, y los que realizan un procesamiento paralelo, es decir, decisiones independientes y simultáneas [6].

La clasificación elegida tiene varios aspectos en común con la última presentada y descrita [8]. Se ha preferido esta clasificación al ser clara en cuanto a la división de las técnicas y las características que éstas tienen frente a otras clasificaciones. En la figura 3.1 se muestra el esquema de la clasificación elegida.

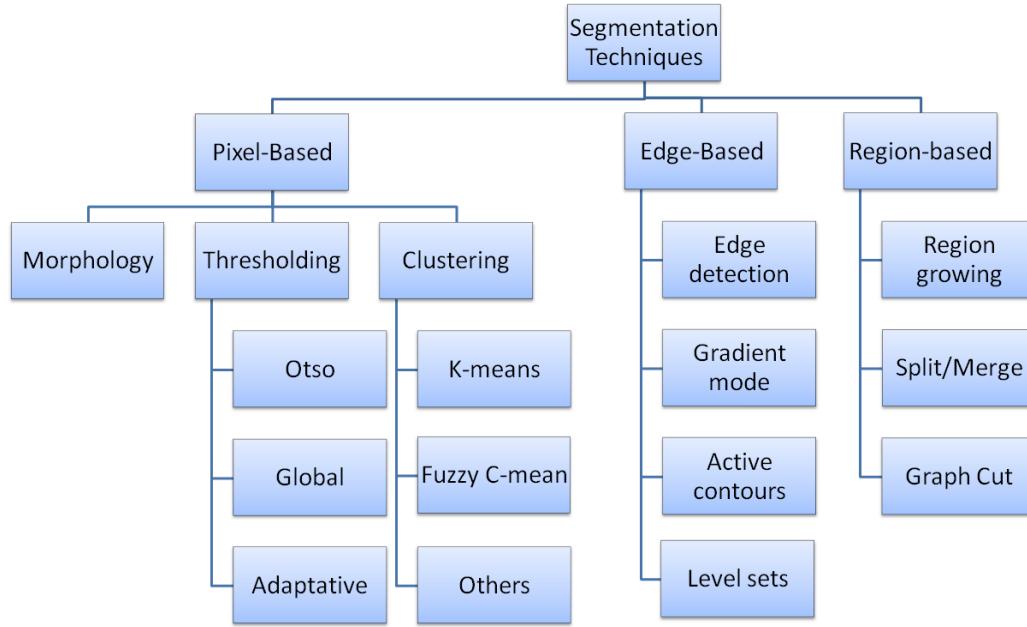


Figura 3.1: Clasificación de las técnicas de segmentación

Fuente: [8]

3.2. Segmentación de imagen basada en el tratamiento de los *píxeles*

Este tipo de segmentación consiste en dividir la imagen en segmentos o conjuntos de píxeles (conocidos como «superpíxeles»). Cada píxel de la imagen será tratado y agrupado en función de sus características. Existen varios subgrupos dentro de esta clasificación: *Thresholding*, o «Método del valor umbral» en castellano, y *Clustering*, o algoritmos de agrupamiento en castellano, entre otros.

3.2.1. Thresholding

Este tipo de segmentación es la más simple de todas y se basa en clasificar los píxeles en dos grupos en función de la intensidad de éstos: los que superan la intensidad umbral definida y los que no la superan. El resultado de esta segmentación sería una imagen binaria (véase la comparación de las figuras 3.3a y 3.3b).

Esta técnica puede ser definida como:

Para una imagen NxM :

for $i = 1, 2, \dots, N$ and $j = 1, 2, \dots, M$

$$f(n) = \begin{cases} 1 & \text{si } I(i, j) \geq T \\ 0 & \text{si } I(i, j) < T \end{cases} \quad (3.1)$$

donde $I(i, j)$ es el valor del píxel de la posición i, j de la imagen.

También existe la posibilidad de definir varias intensidades umbrales (véase la comparación de las figuras 3.4a y 3.4b), con el fin de particionar la imagen en más segmentos. En sí, se podrán definir tantos umbrales como niveles de gris contiene la imagen, aunque habrá que buscar un buen equilibrio.

La ventaja de este tipo de segmentación es que es relativamente sencilla comparada con otros tipos de segmentación más avanzada como *watershed* o *level set*. Aun así, esta segmentación funciona bien cuando el fondo y los objetos siguen una distribución bimodal, es decir, que hay una diferencia «notable» entre las dos partes. Comúnmente esta característica no se da en todas las imágenes, por lo que no tendrá buenos resultados en las que el fondo no se distinga bien de los objetos [8]. Además, esta segmentación tampoco se comporta bien con imágenes que tienen una iluminación gradiente grande (véase la comparación de las figuras 3.5a y 3.5b). Aunque es verdad que para ello también existen varias mejoras que hacen que la segmentación se «adapte» para conseguir mejores resultados.

En la figura 3.1 se muestran varias subclases de *thresholding*. Aunque no se explicarán en profundidad conviene saber que hay varias maneras de realizar esta segmentación. Primeramente se nombra el método de Otsu, *Otsu's method* en inglés, que adapta el umbral en función de la dispersión de los niveles de gris. El segundo método, el *Global*, es el más simple y el que hemos estado explicando hasta ahora. Y por último, el método *Adaptative*, mencionado anteriormente, que se adapta a la intensidad de la imagen.

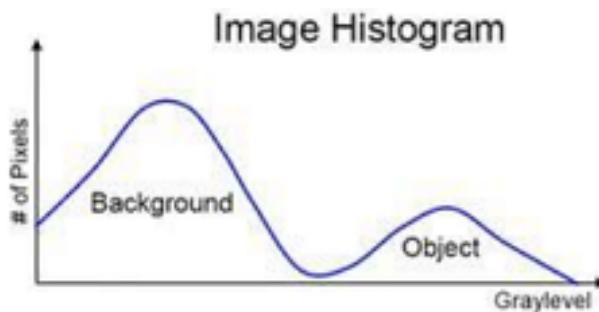


Figura 3.2: Histograma que muestra tres aparentes segmentos de la imagen con dos umbrales

Fuente: [8]

Las figuras 3.3, 3.4 y 3.5 muestran varios ejemplos de segmentación utilizando *thresholding*.

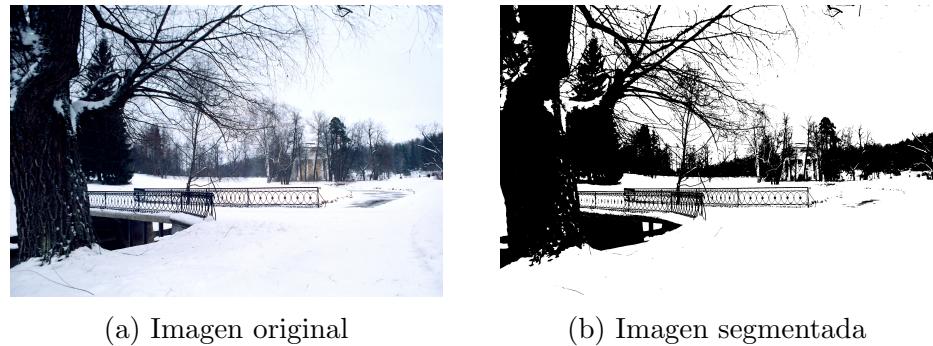


Figura 3.3: Segmentación con *thresholding*

Fuente: commons.wikimedia.org

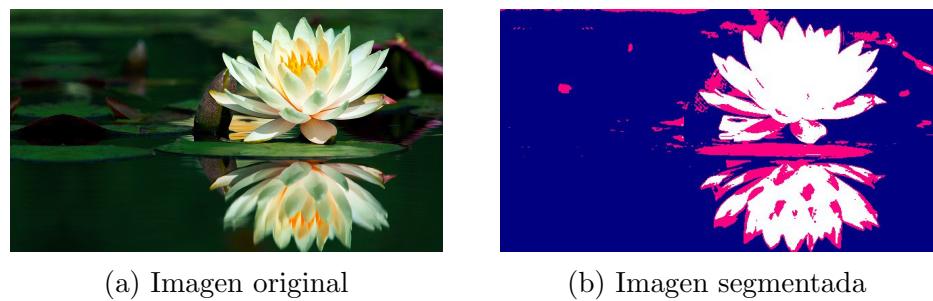


Figura 3.4: Segmentación con *thresholding* con varios umbrales

Fuente: rosavallsformacio.tv y photo-kako.com para la realización de la segmentación

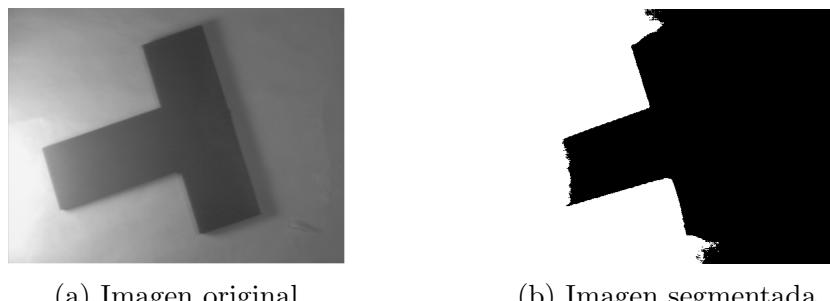


Figura 3.5: Segmentación con *thresholding* en imagen con iluminación gradiente

Fuente: homepages.inf.ed.ac.uk/rbf/HIPR2/

3.2.2. Clustering

Junto con la técnica de *thresholding* el *clustering* es la técnica de segmentación más utilizada. En general esta técnica divide los puntos en varios *clusters* o grupos en función de la distancia entre ellos. En este caso concreto, los puntos serán los píxeles de la imagen y la distancia entre ellos estará relacionada con la intensidad, color y textura, pudiendo combinar varios de estos factores.

Los algoritmos de *clustering* se pueden dividir entre jerárquicos y particionales, donde la principal diferencia entre los dos está en el modo en que se construyen los grupos. Los algoritmos jerárquicos suelen ser más precisos, sin embargo, no valen para una cantidad de datos grande como en una imagen ya que el coste computacional es muy elevado. Por lo tanto, la opción escogida suele ser el *clustering* particional. No obstante, esta técnica tiene varias desventajas [9]:

1. Generalmente es necesario saber previamente el número de *clusters* que hay en la imagen.
2. No utilizan información espacial inherente a la imagen.
3. En algunos algoritmos de clustering, como el K-means que se explicará a continuación, no se asegura un resultado óptimo, ya que distintas inicializaciones dan diferentes resultados.

En la clasificación presentada en la figura 3.1 se muestran varias subclases de la técnica de *clustering* que corresponden a diferentes algoritmos de creación de *clusters*. El algoritmo *Fuzzy C-Means* agrupa los píxeles utilizando la lógica difusa, donde cada píxel tendrá un grado de pertenencia a todos los *clusters*. En general, hay muchos tipos de técnicas de clustering por lo que las técnicas restantes se agruparán en la sección de *Others* de la figura 3.1. El algoritmo K-means es el más utilizado (véase ejemplos de su segmentación en las figuras 3.6 y 3.7). Una descripción de su modo de funcionamiento a grandes rasgos sería:

1. Se asignan K primeros píxeles como centroides.
2. Se agrupan los píxeles, restantes con los centroides definidos en función de la distancia con estos.
3. Se calculan los nuevos K centroides como los baricentros de los K conglomerados obtenidos.

4. Se alternan los pasos 2 y 3 hasta que se alcance un determinado criterio de convergencia (máxima variación de centroides, por ejemplo).

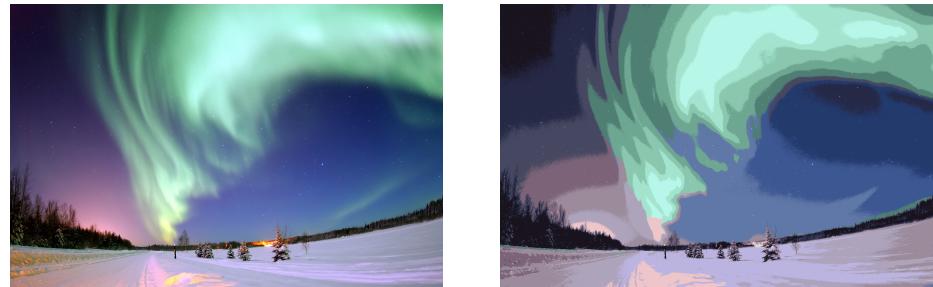


Figura 3.6: Segmentación utilizando K-means con $K=16$

Fuente: commons.wikipedia.org

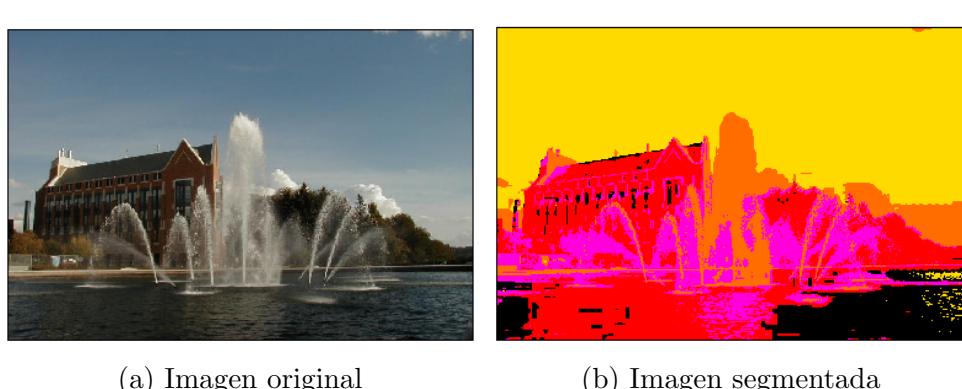


Figura 3.7: Segmentación utilizando K-means con $K=6$

Fuente: imagedatabase.cs.washington.edu/demo/kmcluster/

3.2.3. Morphology

La técnica de la morfología, o *morphology* en inglés, se clasificaría dentro de las técnicas basadas en el tratamiento de los píxeles. Hay varios métodos de morfología y se basan en una máscara llamada *structuring element* para investigar cada píxel. El valor de cada píxel está determinado por el de sus vecinos que pertenecen a esa máscara. Los métodos más simples de esta técnica son la dilatación y la erosión. Para una imagen binaria, la dilatación convierte en uno todos los píxeles de la máscara si los píxeles «debajo» del píxel central son ceros como se muestra en la imagen 3.8. La erosión es el

caso contrario, es decir, convierte en ceros todos los elementos de la máscara si esta contiene algún elemento que sea cero. La combinación de estas simples operaciones junto con otras como el complemento, la unión y la intersección, pueden utilizarse para realizar operaciones más avanzadas y complejas.

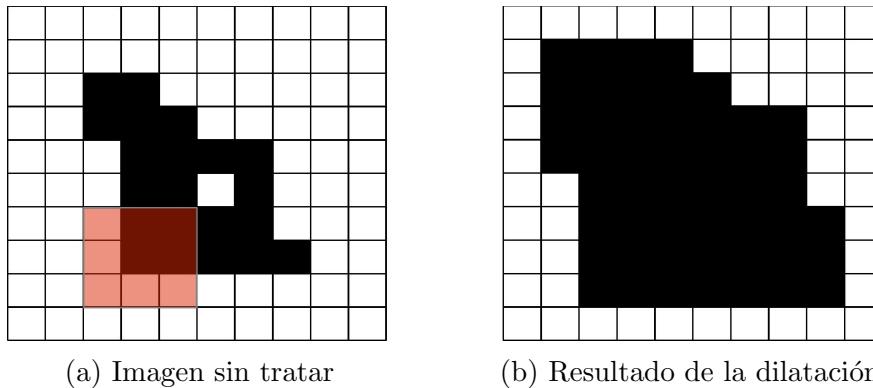


Figura 3.8: Ejemplo de dilatación de la técnica de morfología

Fuente: [10]

3.3. Segmentación de imagen basada en la detección de bordes

Este tipo de segmentación consiste en encontrar los bordes de los objetos contenidos en la imagen con el fin de poder dividir la imagen en función de los bordes encontrados. Los detectores de bordes tradicionales suelen utilizar los operadores diferenciales de detección de bordes comentados en 2.2, es decir, los operadores *Sobel*, *Roberts* y *Prewitt edge detectors* que están basados en el gradiente de la función de intensidad de una imagen. Normalmente los bordes suelen detectarse en las intersecciones de dos regiones de la imagen que tienen diferentes intensidades.

La ventaja de este tipo de segmentación frente a la basada en el tratamiento de píxeles es que, aparte de hacer la división de las diferentes regiones, sabremos exactamente dónde se encuentran los bordes de éstas, siendo útil para poder extraerlas y poder tratarlas individualmente. Por lo tanto, esta técnica funcionará mejor cuando la diferencia entre las regiones tenga buena calidad. Una de las posibles «desventajas» puede ser que la detección de muchos bordes dificulte la extracción de las regiones de interés.

Existen varios subgrupos dentro de esta clasificación: *Edge detection* o «Detección de bordes» en castellano, técnicas que tienen que ver con el gra-

diente de la imagen *Gradient mode*, *Active contours* o «Contornos activos» y *level sets* o técnicas del «Conjunto de nivel».

3.3.1. Detección de borde usando gradientes

En la figura 3.1 se diferencian las dos primeras clasificaciones *Edge detection* y *Gradient mode* pero al estar directamente relacionadas se ha decidido explicarlas en conjunto.

Las técnicas clásicas de detección de bordes se basan en encontrar la derivada respecto a los ejes que forman la imagen, o dicho de otro modo, el gradiente. El gradiente de un punto de una función escalar, representado con ∇ , se representa en forma vectorial. Este vector indica la dirección en la cual la función varía más rápidamente y su módulo representa el ritmo de variación de la función en la dirección de dicho vector. Este módulo se utilizará para determinar si un punto es borde o no, en función de si supera un valor umbral dado. Para encontrar la máxima variación en ese punto se deben de hacer las derivadas parciales respecto a cada eje y coger el máximo valor de éstas. En general el gradiente se suele aproximar con la fórmula¹ $|G| \approx |G_x| + |G_y|$ que es mucho más simple de implementar en la práctica. Valiéndonos de esto, se desarrollaron los primeros operadores diferenciales ya comentados *Sobel*, *Roberts* y *Prewitt edge detectors* (véase un ejemplo de cada segmentación en la figura 3.9). Estos operadores no son más que máscaras aplicadas al píxel a tratar y a cierta vecindad de éste para calcular una aproximación a dichas derivadas G_x y G_y . De ahí el nombrarlos como «operadores».

■ Roberts operator

Este operador es el más simple de los tres mencionados y approxima las derivadas tomando la diferencia de dos valores contiguos. La gran desventaja de este operador es que es muy sensible al ruido al tratar pocos vecinos y sólo permite marcar los puntos del borde pero no su orientación. A pesar de todo ello, es un operador que computacionalmente es poco costoso debido a su simplicidad y que trabaja bien con imágenes binarias. Véase las máscaras utilizadas por esta técnica presentada en la tabla 3.1.

¹Válida para una imagen de dos dimensiones. En caso de tener tres dimensiones la fórmula sería $|G| \approx |G_x| + |G_y| + |G_z|$

$$\begin{array}{|c|c|} \hline +1 & 0 \\ \hline 0 & -1 \\ \hline \end{array}
 \qquad
 \begin{array}{|c|c|} \hline 0 & +1 \\ \hline -1 & 0 \\ \hline \end{array}$$

G_x G_y

Tabla 3.1: Máscaras utilizadas por el operador de Roberts de tamaño 2x2

■ Sobel operator

Este operador utiliza una máscara más grande que el *Roberts operator*, 3x3, por lo que implicará a más vecinos. Enfatiza más los píxeles de alrededor del centro. La ventaja de este operador es que es menos sensible al ruido, detecta muy bien los bordes horizontales y verticales y además proporciona un suavizado. Las desventajas de este operador es que computacionalmente es más costoso, no tiene buena detección de bordes diagonales y no da información sobre la orientación del borde. Véase las máscaras utilizadas por esta técnica presentada en la tabla 3.2.

$$\begin{array}{|c|c|c|} \hline -1 & 0 & +1 \\ \hline -2 & 0 & +2 \\ \hline -1 & 0 & +1 \\ \hline \end{array}
 \qquad
 \begin{array}{|c|c|c|} \hline +1 & +2 & +1 \\ \hline 0 & 0 & 0 \\ \hline -1 & -2 & -1 \\ \hline \end{array}$$

G_x G_y

Tabla 3.2: Máscaras utilizadas por el operador de Sobel de tamaño 3x3

■ Prewitt operator

Este operador es parecido al operador de Sobel pero este no enfatiza los píxeles cercanos al centro y los coeficientes son diferentes. Las ventajas son que aumenta la respuesta a los bordes diagonales poniéndole peso a píxeles vecinos que antes no tenían, tiene poca sensibilidad al ruido y proporciona la magnitud y orientación del borde (hasta 8 direcciones). Véase las máscaras utilizadas por esta técnica presentada en las tablas 3.3 y 3.9.

$$\begin{array}{|c|c|c|} \hline -1 & +1 & +1 \\ \hline -1 & -2 & +2 \\ \hline -1 & +1 & +1 \\ \hline \end{array}
 \qquad
 \begin{array}{|c|c|c|} \hline +1 & +1 & +1 \\ \hline -1 & -2 & +1 \\ \hline -1 & +1 & +1 \\ \hline \end{array}$$

0 45

Tabla 3.3: Máscaras utilizadas por el operador de Prewitt de tamaño 3x3

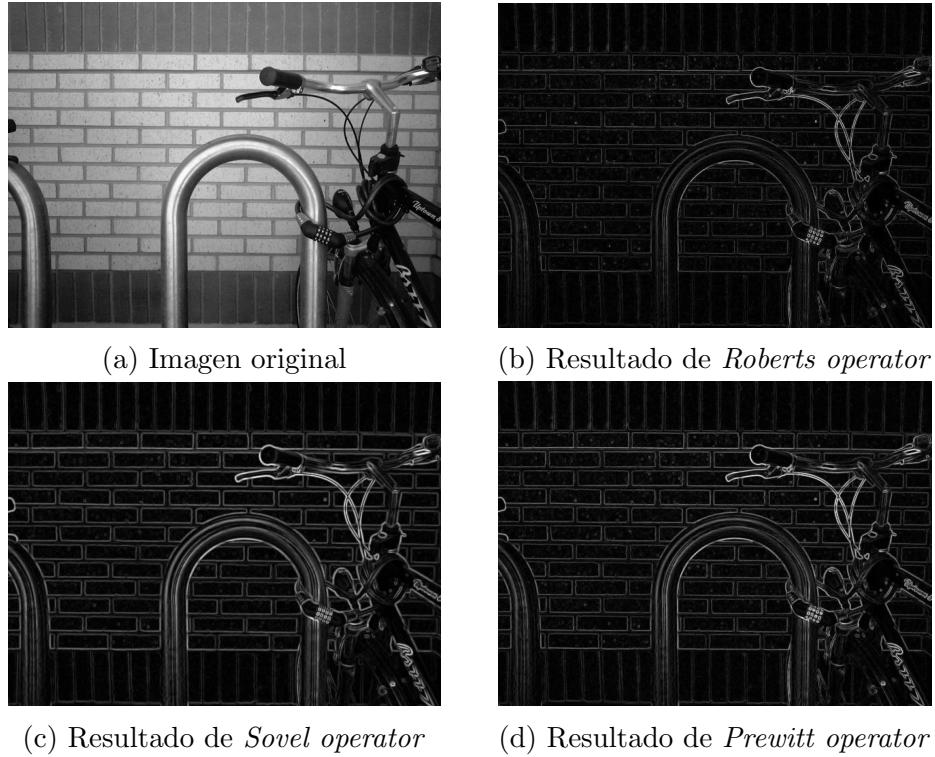


Figura 3.9: Detección de bordes con el uso de los operadores

Fuente: commons.wikipedia.org

3.3.2. Active Contours

Desde que fueron introducidos por Kass y colaboradores en 1988 [11], los contornos activos o más comúnmente nombrados como *Snakes*, han ganado popularidad gracias a los buenos resultados que se pueden llegar a obtener en la segmentación de imágenes.

Un contorno activo o *Snake* es una curva elástica que comienza a moverse dada una posición inicial de manera que llegue a delimitar las regiones de interés de la imagen. La curva se irá moviendo de manera que se minimice su energía hasta llegar a un punto de convergencia. El contorno puede ser definido paramétricamente como $V(s) = [x(s), y(s)]$ donde $x(s)$ e $y(s)$ son las coordenadas de la parte s del contorno. La energía del contorno está compuesta por una energía interna y otra externa, E_{int} y E_{ext} respectivamente. La definición formal sería:

$$E = \int_0^1 E_{int}(v, s) + E_{ext}(v(s)) ds \quad (3.2)$$

E_{int} da las características de deformación del contorno elástico, por lo tanto depende de la forma que éste tenga. La E_{int} puede ser definida como:

$$E_{int} = \frac{1}{2}(\alpha|\frac{\delta v}{\delta s}|^2 + \beta|\frac{\delta^2 v}{\delta s^2}|^2) \quad (3.3)$$

y los valores α y β determinan el grado en el que el contorno se puede estirar o curvar. Un aumento en la magnitud α incrementaría la tensión de la curva y un aumento de β incrementaría la rigidez de la curva, haciendo que sea menos flexible.

En cuanto a la energía externa hay varias maneras de definirla. Una elección popular sería la magnitud negativa del gradiente de la imagen que se definiría como:

$$E_{ext}(\vec{x}) = -|\nabla[G_\alpha I(\vec{x})]|^2 \quad (3.4)$$

donde G_α es una convolución con un filtro pasa bajo gaussiano. Esta propuesta de energía hace que el contorno se expanda hasta los bordes que haya en la imagen como se puede ver en la figura 3.10.

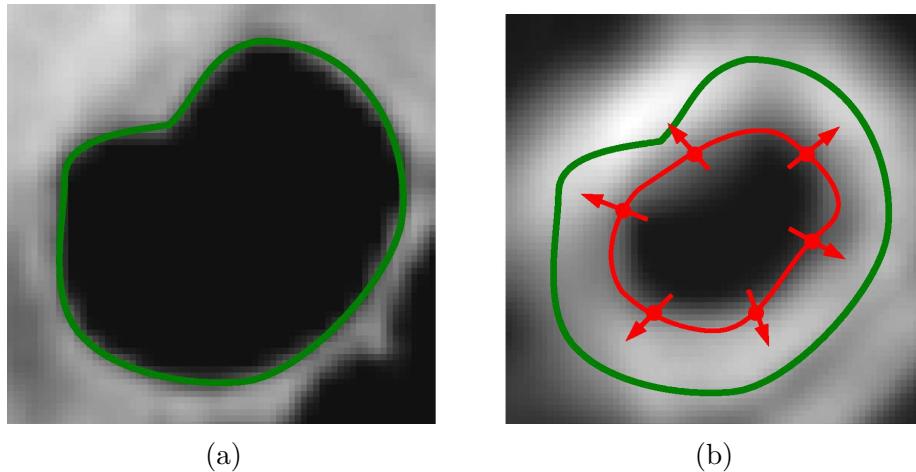


Figura 3.10: Ejemplo de un contorno activo

Fuente: [10]

La imagen 3.10a corresponde a la imagen de entrada, mientras que la imagen 3.10b corresponde a la convolución de la magnitud gradiente de la imagen de entrada con un filtro pasa bajo gaussiano. La línea interior (roja) en la imagen 3.10b es el contorno activo que se moverá hasta la línea exterior (verde) que corresponde con el borde de la imagen original. Un ejemplo de la aplicación de esta técnica se puede ver en la figura 3.11.



Figura 3.11: Ejemplo de segmentación de plantas presentando en [2] utilizando los contornos activos

Fuente: [2]

3.3.3. *Level set*

Conjuntos de nivel, o *level set* en inglés, es un tipo de segmentación muy parecida a la que hemos explicado en la sección 3.3.2 ya que también se trata de expandir un contorno dado previamente para encontrar los bordes de la imagen. La ventaja de este método frente al referido es que permite juntar y dividir contornos sin ningún cálculo extra necesario.

El contorno está representado por la función de *level set*, que está definida en una dimensión más que las dimensiones de la imagen a segmentar, es decir, para una imagen 2D tendríamos una superficie de *level set* de tres dimensiones mientras que para una imagen de 3D tendríamos una superficie de 4D. En el caso de una superficie de 3D ésta tiene una forma cónica como se puede ver en la figura 3.12. Suponiendo entonces que la imagen a tratar tiene dos dimensiones, podríamos definir la función de *level set* como $z = \phi(x, y, t)$ que devuelve la altura de la superficie de *level set* en el punto (x, y) del plano de la imagen en el tiempo t . El contorno es definido implícitamente como «zero *level set*», donde la altura del plano respecto a la superficie es cero ($\phi(x, y, t) = 0$). Esto es justo la intersección entre el plano de la imagen y la superficie. Véase las figuras 3.12 y 3.13 para ver ilustraciones de la técnica de *level set*.

Para propagar el contorno se mueve la superficie de *level set* en el eje z, como se puede ver en la figura 3.12. La rapidez y la dirección en la que se mueve el contorno está determinada por como se curva la superficie de *level set*. Suponiendo que cada punto del contorno se mueve en una dirección ortogonal frente al contorno con una velocidad F, el contorno evoluciona usando la siguiente PDE (*partial differential equation* o ecuación en derivadas parciales en castellano):

$$\frac{\delta\phi(x, y, t)}{\delta t} = F(x, y, I)|\nabla\phi(x, y, t)| \quad (3.5)$$

Esta función de velocidad varía dependiendo del punto de la imagen I a tratar y hace que el contorno se expanda a ciertas áreas de la imagen y no lo haga en otras zonas de esta. Normalmente la función de velocidad se define por la intensidad o el gradiente de los píxeles y por la curva de la función de *level set*.

De la idea de que modificaciones de píxeles lejanos al contorno no afectan a este surgen varias mejoras de esta técnica que tienen en cuenta los píxeles con los que se trabajará en cada iteración: *narrow band* y *sparse field methods*. El método de *narrow band* actualiza los píxeles en una línea estrecha alrededor del contorno. El método *sparse field* actualiza los píxeles vecinos del contorno únicamente.

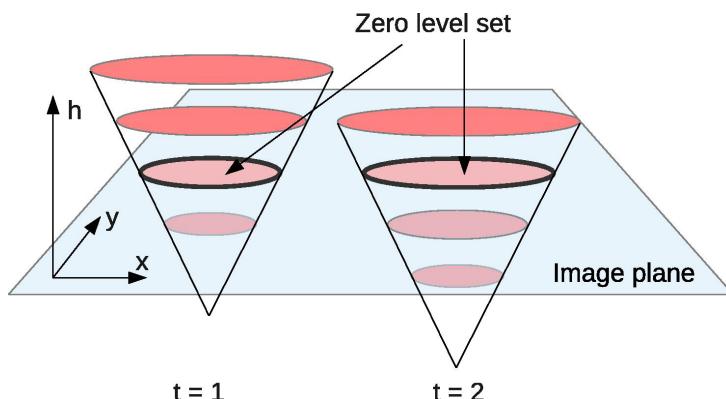


Figura 3.12: Ilustración de la superficie de *level set*

Fuente: [10]

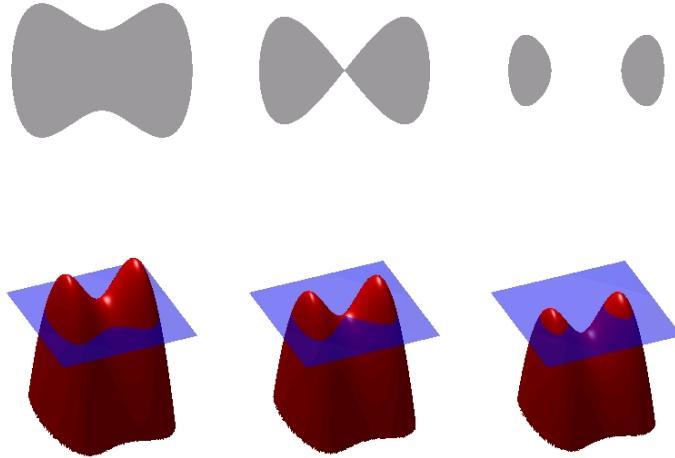


Figura 3.13: Ilustración del método de *level set*

Fuente: commons.wikipedia.org

3.4. Segmentación de imagen basada en regiones crecientes

Este tipo de segmentación, más conocida como *region growing*, se basa en la idea de que los píxeles de una región tienen características comunes, como puede ser, la intensidad de gris etc. Por ello, al tener que tratar un nuevo píxel si este tiene una intensidad de gris parecida a la intensidad de grises que contiene la región significará que ese punto pertenece a ella.

Existen varios subgrupos dentro de esta clasificación: técnicas basadas en la idea de regiones crecientes, o *region growing* en inglés, técnicas de *Split/Merge* y técnicas basadas en grafos *Graphs cuts*.

3.4.1. *Region growing*

Este subgrupo agrupa todos las técnicas que siguen la idea de agrupar los píxeles con características comunes, como el nivel de gris o el color de éstos. Hay dos técnicas conocidas que siguen esta metodología, por lo que en esta ocasión explicaremos más de una técnica de este subgrupo de segmentación.

■ ***Region growing o Seeded-based region growing segmentation***

Este tipo de segmentación comienza con la selección de un píxel, denominado a menudo como semilla o *seed* en inglés, que está dentro del objeto de interés. Normalmente la semilla se elige manualmente. A partir de ese píxel semilla (primer punto de la región) se comenzará a extender la región procesando sus vecinos y añadiéndolos en base a un criterio predefinido. Este criterio de inserción será en base a la intensidad, color o textura de la semilla y los puntos que pertenezcan a la región. Cada vez que se inserta un nuevo punto a la región la característica que se esté utilizando para realizar la inserción se volverá a calcular, por ejemplo, si se utiliza el nivel de gris, se volverá a calcular el valor medio de los niveles de gris que hay en la región generada hasta el momento. De esta manera, la región se irá expandiendo añadiendo vecinos hasta que encuentre alguno que no cumpla con la condición de inserción impuesta por el criterio. Si un punto no ha sido añadido a ninguna región se podrá añadir a una región cercana suya si la diferencia entre, por ejemplo, el nivel de gris de este punto y el nivel de gris medio de la región no supera un valor umbral T dado. Véase un ejemplo de esta técnica en la figura 3.14.

Esta técnica es útil cuando la intensidad del fondo y del objeto son muy parecidas pero están separadas por un borde «notable» o por otra región.

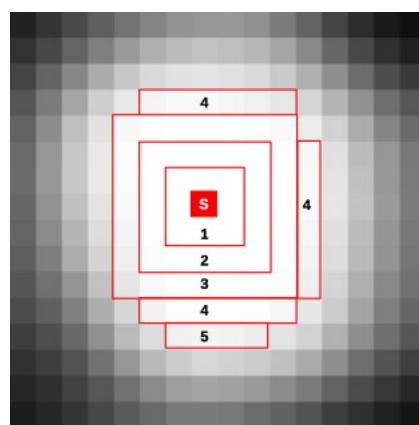
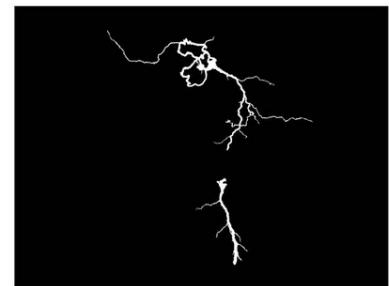
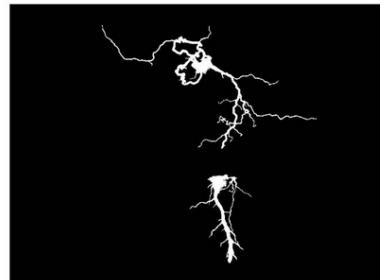
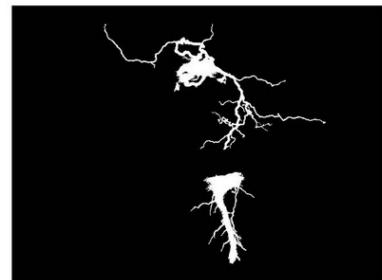


Figura 3.14: Ejemplo del funcionamiento de la técnica de *region growing*

Fuente: [10]



(a) Imagen original

(b) *Region growing* con un valor umbral $T=255$ (c) *Region growing* con un valor umbral $T=225$ (d) *Region growing* con un valor umbral $T=190$ (e) *Region growing* con un valor umbral $T=155$ Figura 3.15: Ejemplo de la técnica de *region growing*

Fuente: commons.wikipedia.org

En la figura 3.15 se puede ver un ejemplo de la técnica de *region growing* donde se quiere encontrar la parte del rayo más fuerte en la imagen. Para ello se eligen como semilla los puntos con mayor valor de gris posible(255). El criterio de inserción de los píxeles es tener el "mismo" nivel de gris. En este caso se ha decidido que los puntos que no han sido insertados en una región

se inserten en una cercana si superan el valor umbral T dado. Por ello en 3.15b solo aparecen los puntos semilla ya que no se habrán unido puntos con este criterio y el valor T es demasiado alto. En las imágenes 3.15c, 3.15d y 3.15e ese valor T se va disminuyendo y cada vez se insertan más puntos en la región.

■ **Watershed**

La idea principal de esta técnica se basa en ver la imagen como una imagen tridimensional donde la tercera dimensión es la altura del píxel. La altura del píxel está determinada por su nivel de gris. Topológicamente quedará algo como se puede ver en 3.16. En este «terreno» creado se podrán diferenciar hasta tres puntos. Estos puntos son determinados con la analogía de como una gota de agua caería y se movería si se precipitase en ese punto. Hay tres tipos de puntos:

1. Puntos con un nivel de gris mínimo local donde la gota se estancaría.
2. Puntos en los que la gota caería o se deslizaría hacia un único punto mínimo local.
3. Puntos en los que la gota podría caer hacia más de un punto mínimo local.

El segundo tipo de puntos son nombrados como *watershed*, o cuencas en castellano, y los del tercer tipo son nombrados como *watersheds lines*, bordes o líneas de las cuencas en castellano.

El objetivo final de esta técnica será encontrar los bordes de las cuencas, que representarán los bordes de la imagen original. Para ello existen varias maneras de hacerlo, la más común es la técnica de *flooding*. La idea es simple, imaginemos que se empieza a echar agua en los puntos de tipo uno, es decir, los que representan un mínimo local y son «cuencas». El nivel del agua empezará a subir hasta que llegue a un punto en el que la cuenca se empiece a desbordar y vaya a juntarse con otra cuenca. En ese momento, se construye una presa o un muro de manera que el agua no se desborde. Esas presas o muros construidos serán los bordes de las cuencas. Con todas estas presas se habrá conseguido la segmentación de la imagen.

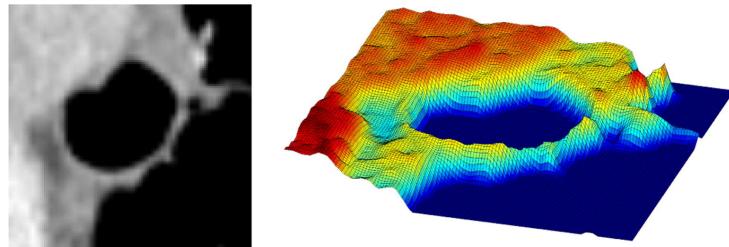


Figura 3.16: Ejemplo del funcionamiento de la técnica de *watershed*

Fuente: [10]

En la figura 3.17 se puede ver el un ejemplo del funcionamiento de la técnica de *watershed* donde la intensidad de los píxeles de la imagen de la izquierda serán la altura de ese mismo punto en la imagen derecha, creando así ese terreno.

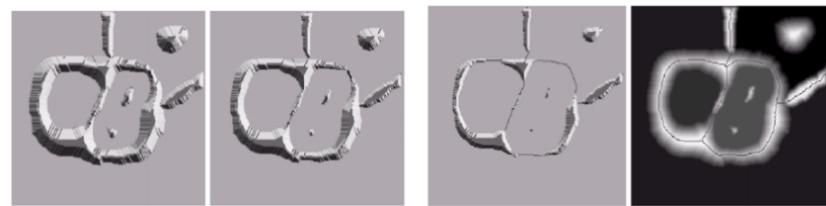


Figura 3.17: Funcionamiento de la técnica de *watershed*

Fuente:

<http://www.di.ubi.pt/~agomes/cvm/teoricas/07-regionsegmentation.pdf>

En la figura 3.18 se muestra el funcionamiento de la técnica de *watershed* donde se puede ver como va creciendo el nivel de agua hasta encontrar ese «punto» de desbordamiento en el que se marcan los bordes.

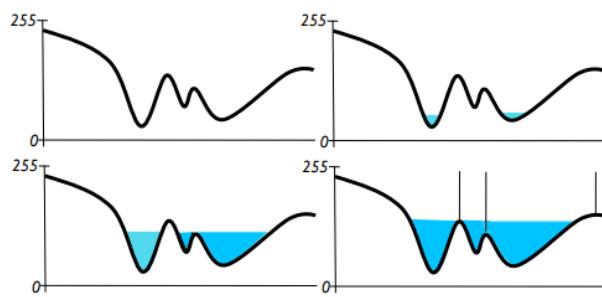


Figura 3.18: Funcionamiento de la técnica de *watershed*

Fuente:

<http://www.di.ubi.pt/~agomes/cvm/teoricas/07-regionsegmentation.pdf>

3.4.2. *Split/Merge*

La técnica de *split*, a diferencia de la técnica de *region growing* que empieza con una serie de puntos «semilla», empieza con la imagen entera como una única región y va subdividiendo la imagen recursivamente en regiones más pequeñas en base a un criterio de homogeneidad.

En cuanto a la técnica de *merge*, es lo contrario que la de *split*, ya que esta empieza con pequeñas regiones de 2x2 o 4x4 y las va juntando entre sí en base a si tienen o no características comunes entre ellas como el nivel de gris, el color, la textura etc.

3.4.3. *Graph cut*

La idea principal de este tipo de segmentación es representar la imagen a tratar como un grafo, donde normalmente cada píxel es un nodo y tienen aristas con los nodos vecinos. La ventaja de estos algoritmos está en que pueden llegar a trabajar bien incluso si la separación entre dos regiones está «rota» o es dudosa. Hay varios algoritmos distintos dentro de esta clasificación, en este caso se ha seleccionado el método de segmentación de *Markov*, o *Markov random field (MRF) segmentation* y concretamente una variante de este llamada *graph cut*.

MRF considera a cada píxel como un nodo del grafo y tienen aristas con cada píxel vecino. Sin embargo, cada nodo tiene dos conexiones más a un par de nodos especiales llamados *source* (S) y *sink* (T) como se muestra en la figura 3.19. Se les añade un peso a las aristas entre los nodos de manera que los píxeles que pertenecen al fondo tienen un peso pequeño en la arista que los une con uno de esos dos nodos anteriores y un peso grande con el otro nodo. De forma inversa, los píxeles que pertenezcan al primer plano tendrán un peso pequeño con uno de ellos y un peso grande con el otro. Por otro lado, los pesos de las aristas entre los píxeles son grandes cuando éstos tienen características comunes y un peso pequeño en caso contrario.

La segmentación se realiza aplicando un algoritmo de corte de grafos. El objetivo es minimizar la suma de las aristas por las que se va a cortar el grafo, para ello hay varios algoritmos para buscar el mínimo corte ha realizar.

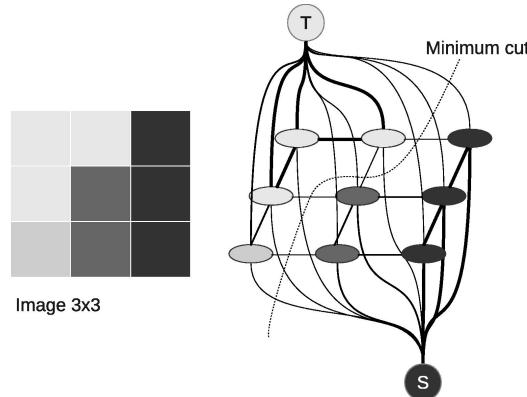


Figura 3.19: Funcionamiento de la técnica de *graph cut* en una imagen 3x3

Fuente: [10]

3.5. Conclusiones de los tipos de segmentación

Como se ha podido ver a lo largo de este capítulo, hay muchos tipos de técnicas de segmentación de imágenes. Además, existen más técnicas de segmentación que no se han explicado en este trabajo ya que se han añadido las más conocidas y populares con el fin de establecer una base para el lector.

A parte de las técnicas que no se han explicado hay que decir que existen muchas mejoras, optimizaciones y variantes de todas las técnicas de segmentación. Por ello, en este trabajo se han explicado las ideas principales de los algoritmos sin tener en cuenta ésto ya que el trabajo se habría extendido demasiado.

Capítulo 4

Optimizaciones y mejoras del algoritmo *level set* original

El algoritmo *level set* es uno de los más utilizados como consecuencia del buen rendimiento que obtiene al segmentar imágenes. Existen optimizaciones que lo hacen muy eficiente, como el trabajo que se presentará a continuación [12], que será capaz de satisfacer la demanda que requiere este proyecto.

4.1. Introducción

Como se ha dado a conocer en el apartado 3.5 hay muchos tipos de mejoras y optimizaciones de cada técnica de segmentación existentes. A la hora de afrontar el problema que el cliente proponía se tenía que elegir un tipo de segmentación de las técnicas estudiadas. Entre todas ellas se escogió el algoritmo *level set*, por ser un algoritmo de segmentación eficiente ampliamente utilizado, lo que facilitaría la búsqueda de información sobre él. Además, el cliente recomendaba esta técnica, ya que aseguraba que se iban a obtener buenos resultados con ella.

Repasando este algoritmo, las ventajas que tiene son que la segmentación conseguida es precisa y que no añade sobrecoste al hacer la división o unión del contorno. Sin embargo, el algoritmo se basa en la resolución de ecuaciones diferenciales, lo que supone que el algoritmo sea costoso y lento. Se han desarrollado varias implementaciones en GPU [10], también sobre imágenes 3D [13], y en proyectos de fin de carrera también se utilizan GPUs [14] etc. De la misma manera, también se han desarrollado paralelizaciones del algoritmo para arquitecturas SMP(*Symmetric Multi-Processing*) [15] o para

arquitecturas de memoria distribuida con MPI(*Message Passing Interface*) [16].

Hay muchos artículos sobre este algoritmo, alguna implementación paralela realizada sobre el algoritmo original y, por lo tanto, a pesar de la paralelización, los tiempos que obtienen siguen siendo elevados para las restricciones de este proyecto. Estas parallelizaciones y otras técnicas de optimización se centran siempre en resolver las PDEs asociadas a la evolución del *level set*. Sin embargo, para muchos problemas de imagen, como la segmentación, no es necesario tanta precisión, ya que el objetivo final es encontrar los bordes de los objetos. En este caso, la evolución del proceso no tiene tanto interés como el resultado final. Siguiendo esta idea Shi y Karl presentaron un artículo muy interesante que será la base de la segmentación realizada en este proyecto [12].

4.2. Aproximación a la técnica de *level set*

Como se ha mencionado anteriormente para el objetivo de este proyecto, no se tienen por qué resolver las PDEs en el algoritmo *level set* ya que, para una segmentación, importa más el resultado final que la evolución propia del algoritmo. Además, la resolución de las PDEs conlleva que se tengan que realizar reinicializaciones de la función de *level set* (representada con la letra ϕ) lo que implica aún más cálculo. El trabajo presentado por Shi y Karl [12] elimina la reinicialización al tener una colección de enteros (los cuales representan la función ϕ) que cambian dinámicamente según va propagándose el contorno y, además, no calculan las PDEs. Estas dos mejoras elementales hace que su algoritmo sea mucho más rápido que el *level set* original.

En este trabajo se presenta una nueva estrategia de implementación del método de *level set*. Para el caso de un espacio euclídeo de dos dimensiones, una curva C es representada implícitamente como el *zero level set* de una función ϕ definida en una cuadrícula como se muestra en la figura 4.1. La función ϕ tendrá un valor negativo dentro de la curva C y un valor positivo fuera de ella, por eso se dice que representa implícitamente a la curva C.

Se definen dos listas de vecindad en esta cuadrícula, L_{in} y L_{out} . En la imagen se puede ver que el movimiento de la curva C se puede conseguir moviendo un punto de una lista a otra.

Se asume que la función ϕ está definida sobre un dominio $D \subset R^k$ y está discretizada sobre una rejilla de tamaño $M_1 \times M_2 \times \dots \times M_k$. Estando entonces en una representación de dos dimensiones, y siguiendo el ejemplo de la rejilla mostrada anteriormente se pueden definir dos listas de vecindad

del contorno C: L_{in} y L_{out} .

$$L_{in} = \{x \mid \phi(x) < 0 \text{ y } \exists y \in N(x) \text{ tal que } \phi(y) > 0\}$$

$$L_{out} = \{x \mid \phi(x) > 0 \text{ y } \exists y \in N(x) \text{ tal que } \phi(y) < 0\}$$

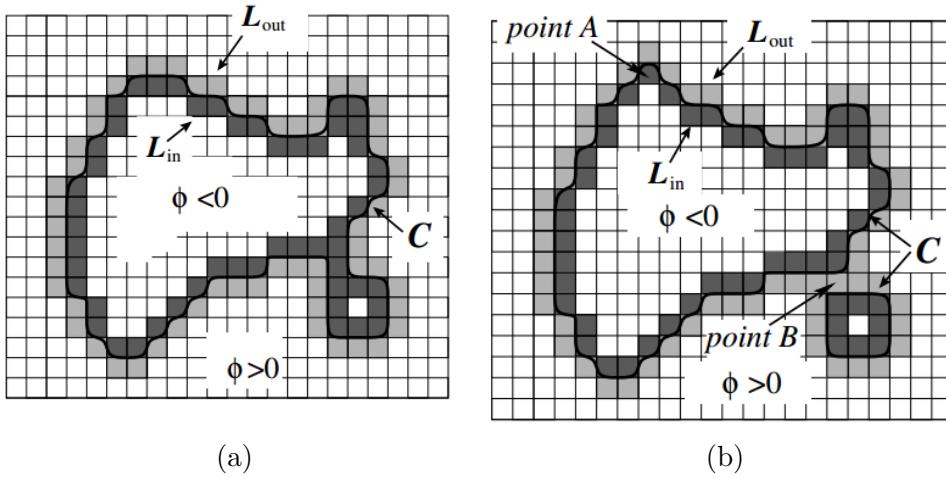


Figura 4.1: Representación de la curva C

Fuente: [12]

Siendo x una coordenada de la rejilla denotada como $x = x_1, x_2, \dots, x_k$ y $N(x)$ como un vecino de x con valor discreto definido como:

$$N(x) = \{y \in D \mid \sum_{k=1}^K |y_k - x_k| = 1\} \quad \forall x \in D \quad (4.1)$$

Como podemos observar en la figura 4.1 la lista L_{in} está formada por los puntos de la rejilla que están dentro de la curva C y L_{out} está formada por los puntos de la rejilla que están fuera de C. Por lo tanto, como se puede ver en las definiciones formales de las listas, cada punto de ellas tiene que tener un punto vecino de la otra lista, de manera que las dos estén "pegadas".

Recordando lo visto en la sección de explicación del *level set* (3.3.3) en el clásico *level set* la siguiente PDE es resuelta para evolucionar el contorno C bajo una función de velocidad F:

$$\frac{d\phi}{dt} + F|\nabla\phi| = 0 \quad (4.2)$$

La figura 4.1b muestra el proceso evolutivo de la curva C de la figura 4.1a. En el punto marcado como A la curva se ha movido hacia fuera lo que ha modificado el valor de la función ϕ de positivo a negativo. En el punto B la curva se ha movido hacia dentro, partiendo la curva en dos y cambiando el valor de la función ϕ de negativo a positivo. Todo esto también ocurre en el *level set* original, con la diferencia de que la resolución de la PDE mostrada en la ecuación 4.2 tiene un coste elevado. Se consiguen los mismos resultados finales fácilmente si se usa la relación entre C, L_{in} y L_{out} . Para mover la curva hacia fuera en el punto A de la rejilla tendremos que pasar el punto de la lista L_{out} a L_{in} . De manera similar, para mover la curva hacia dentro en el punto B tendremos que cambiar el punto B de L_{in} a L_{out} . En general, con aplicar dichas operaciones se va moviendo la curva hacia cualquier punto con el mínimo coste operacional.

Algoritmo

Para la realización del algoritmo son necesarias estas estructuras:

- Un array para la función de *level set* ϕ ;
- Un array para la velocidad (F) con la que se propagará la curva;
- Dos listas de vecindad de la curva: L_{in} y L_{out} .

Nombraremos a los puntos que están dentro de C pero no en la lista L_{in} como «puntos interiores» y a los que están fuera de C pero que no pertenecen a L_{out} como «puntos exteriores». Para agilizar aún más el cálculo, los valores que puede tomar la función ϕ son cuatro enteros: -3, -1, 1, 3.

$$\phi(x) = \begin{cases} 3 & \text{si } x \text{ es un punto exterior} \\ 1 & \text{si } x \in L_{out} \\ -1 & \text{si } x \in L_{in} \\ -3 & \text{si } x \text{ es un punto interior} \end{cases} \quad (4.3)$$

Para la función F de velocidad sólo se usa el signo, por lo que también es un *array* de enteros con los valores: 1, 0, -1. En cuanto a las listas, son listas ligadas de manera que la inserción y el borrado de puntos se pueden hacer de manera rápida.

Antes de presentar el algoritmo se aclararán ciertas cuestiones. Para empezar, hay dos operaciones básicas que se utilizan en el algoritmo:

- La operación *switch_in()* para un punto $x \in L_{out}$ se define como: *switch_in(x)*:

- Paso 1: Se quita el punto de L_{out} y se pasa a L_{in} . Se cambia el valor de ϕ en ese punto: $\phi(x) = -1$.
- Paso 2: Se añaden los puntos vecinos de x a L_{out} y se cambian sus respectivos valores en ϕ . Más formalmente: $\forall y \in N(x)$ que satisfaga $\phi(y) = 3$ se añade y a L_{out} y se pone $\phi(y) = 1$

Con esta operación se mueve el contorno un punto de la rejilla hacia fuera.

- Similarmente se define la operación *switch_out()* para un punto $x \in L_{in}$: *switch_out(x)*:
 - Paso 1: Se quita el punto de L_{in} y se pasa a L_{out} . Se cambia el valor de ϕ en ese punto: $\phi(x) = 1$.
 - Paso 2: Se añaden los puntos vecinos de x a L_{in} y se cambia sus respectivos valores en ϕ . Más formalmente: $\forall y \in N(x)$ que satisfaga $\phi(y) = -3$ se añade y a L_{out} y se pone $\phi(y) = -1$

Con esta operación se mueve el contorno un punto de la rejilla hacia dentro.

Por otro lado, la función F de velocidad presentada antes, normalmente se suele separar en dos velocidades: F_{ext} que depende de los datos y F_{int} para la realización de un suavizado del contorno. La velocidad F_{int} es normalmente la curvatura de la curva [17]. Sin embargo, esta evaluación de la curva usando la función ϕ suele ser computacionalmente costosa. Tras varios pasos esta función de velocidad F_{int} puede llegar a ser determinada por un filtro Gaussiano que puede ser aproximado con operaciones con enteros, por lo que se reduce el cómputo. Además, al separar las dos velocidades, no tiene por qué hacerse el suavizado después de cada iteración de evolución del contorno como en otros trabajos [12], sino que se realizará cuando se satisfaga cierta condición, lo que reduce aún más el coste. Con esto entonces, el algoritmo tendrá dos ciclos principales: el primer ciclo en el que se expandirá o evolucionará el contorno y el segundo ciclo (que se realizará de vez en cuando) en el que se suavizará el contorno para que se siga expandiendo con normalidad.

- Paso 1: Inicializar el «array» ϕ , F_{ext} y las dos listas L_{out} y L_{in} .
- Paso 2: Primer ciclo donde se escanean las dos listas para actualizar ϕ , L_{out} y L_{in} .
 - Se calcula la velocidad para cada punto de L_{out} y L_{in} .
 - Evolución hacia fuera. Recorremos la lista L_{out} y se hace la operación $switch_in(x) \forall x \in L_{out}$ si $F_{ext}(x) > 0$.
 - Se eliminan los puntos redundantes en L_{in} (véase la figura 4.2). Para ello se tendrá que recorrer la lista y para cada punto $x \in L_{in}$, si $\forall y \in N(x); \phi(y) < 0$, se borra x de L_{in} y se cambia $\phi(x) = -3$.
 - Evolución hacia dentro. Recorremos la lista L_{in} y se hace la operación $switch_out(x) \forall x \in L_{in}$ si $F(x) < 0$.
 - Se eliminan los puntos redundantes en L_{out} . Para ello se tendrá que recorrer la lista y para cada punto $x \in L_{out}$, si $\forall y \in N(x); \phi(y) > 0$, se borra x de L_{out} y se cambia $\phi(x) = 3$.
 - Se comprueba la condición de parada y si se satisface se continua al paso 3, si no, seguiremos en el paso 2.
- Paso 3: Segundo ciclo donde se realiza un suavizado del contorno con un filtro Gaussiano.
 - Evolución hacia fuera. Recorremos la lista L_{out} y se calcula $G \oplus \phi(X)$. Si $G \oplus \phi(X) < 0$ se realiza la operación $switch_in(x)$
 - Se eliminan los puntos redundantes en L_{in} (véase la figura 4.2). Para ello se tendrá que recorrer la lista y para cada punto $x \in L_{in}$, si $\forall y \in N(x); \phi(y) < 0$, se borra x de L_{in} y se cambia $\phi(x) = -3$.
 - Evolución hacia dentro. Recorremos la lista L_{in} y se calcula $G \oplus \phi(X)$. Si $G \oplus \phi(X) > 0$ se realiza la operación $switch_out(x)$
 - Se eliminan los puntos redundantes en L_{out} . Para ello se tendrá que recorrer la lista y para cada punto $x \in L_{out}$, si $\forall y \in N(x); \phi(y) > 0$, se borra x de L_{out} y se cambia $\phi(x) = 3$.
- Paso 4: Si se satisface la condición de parada del ciclo uno, se termina el algoritmo, si no, se vuelve al paso 2.

Tabla 4.1: Algoritmo completo de la aproximación del *level set*

Fuente: [12]

Aclaradas las cuestiones anteriores, el (rápido) algoritmo *level set* [12] se muestra en la tabla 4.1. Como se puede observar el segundo ciclo es prácticamente igual al primer ciclo, a excepción de que la condición de cambiar un punto de una lista a otra, es decir, de hacer una de las operaciones *switch()* que se han explicado anteriormente, depende de un filtro Gaussiano (G) en el segundo ciclo (velocidad F_{int}) y de los datos en el primer ciclo (velocidad F_{ext}). Las operaciones de *switch_in()* se realizarán cuando el valor de F (F_{ext} en el primer ciclo y F_{int} en el segundo ciclo) sea positivo, que querrá decir que el contorno deberá expandirse hacia fuera y las operaciones de *switch_out()*

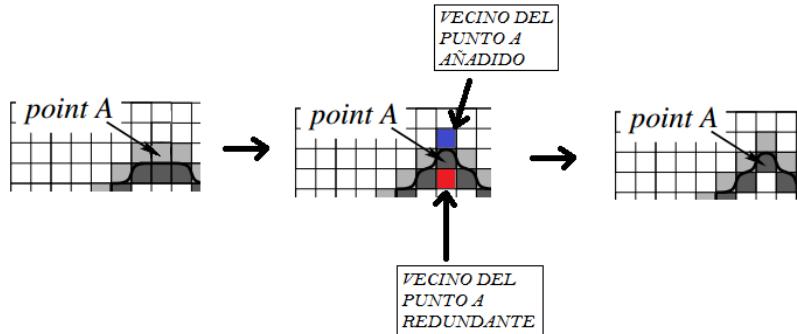


Figura 4.2: Ejemplo de expansión del contorno un píxel hacia fuera.
Operaciones que se realizan continuamente en el paso 2 y 3 del algoritmo

Fuente: modificaciones de imágenes en [12]

se realizarán cuando el valor de F (F_{ext} en el primer ciclo y F_{int} en el segundo ciclo) sea negativo, contrayendo el contorno en ese punto. Véase la figura 4.2 como ejemplo de expansión del contorno en un píxel.

Repasando un poco más el algoritmo 4.1 queda por aclarar las condiciones de parada del algoritmo. Así pues, el algoritmo se parará en el caso de que se cumpla una de estas dos condiciones:

1. Que la velocidad F_{ext} de toda la vecindad del contorno cumpla con:
 - a) $F(x) \leq 0 \forall x \in L_{out}$, es decir, que no se tenga que expandir el contorno por ninguno de sus puntos.
 - b) $F(x) \geq 0 \forall x \in L_{in}$, es decir, que no se tenga que contraer el contorno en ninguno de sus puntos.
2. Que se alcance un determinado número de iteraciones establecido.

Para finalizar, el coste del algoritmo es de orden $O(2A(P_1 + P_2))$, donde A es el número de puntos entre el primer contorno creado y el último ya evolucionado, P_1 y P_2 el coste de las operaciones de `switch()` y el coste que tiene el borrado y la inserción de los puntos en las listas respectivamente. El escalar 2 es debido a que los puntos pueden llegar a pertenecer a una lista primero, y luego a otra, realizando así las operaciones dos veces. Nótese que ese A será mucho menor que la $m \times n$ que sería la anchura (m) por la altura (n) de la imagen, ya que si en la imagen se detectan varios objetos, el tamaño en puntos o píxeles de cada objeto se restará a ese $m \times n$.

4.3. Evolución del contorno

Se ha querido añadir esta sección para destacar y hablar un poco más en detalle sobre la evolución de la curva que se ha elegido en esta aproximación: la presentada en [17]. Este popular trabajo es conocido como *Chan-Vese model* y a diferencia de como se realizaba la segmentación, utilizando el uso de detectores de bordes basados en el gradiente, este trabajo realiza la detección de objetos cuyos bordes no tienen porque estar definidos por el gradiente. Este método es flexible y potente, capaz de realizar la segmentación de muchos tipos de imágenes que con métodos tradicionales como *thresholding* o métodos basados en el gradiente no podían realizarse. Este método también permite que el contorno se inicialice dentro del objeto o incluso entre el objeto y el fondo ya que el resultado final será siempre el mismo. La figura 4.3 muestra las posibles inicializaciones. Esta característica será de utilidad como se comentará en posteriores capítulos.

Para tener una mínima idea del avance que esto supuso en la segmentación de imágenes, si volvemos a utilizar el buscador [1] y encontramos dicho artículo, el número de citaciones de otros artículos registrados en ese explorador alcanza casi la cifra de 2.000, lo que tampoco quiere decir que no haya más trabajos no registrados en [1] que lo citen.

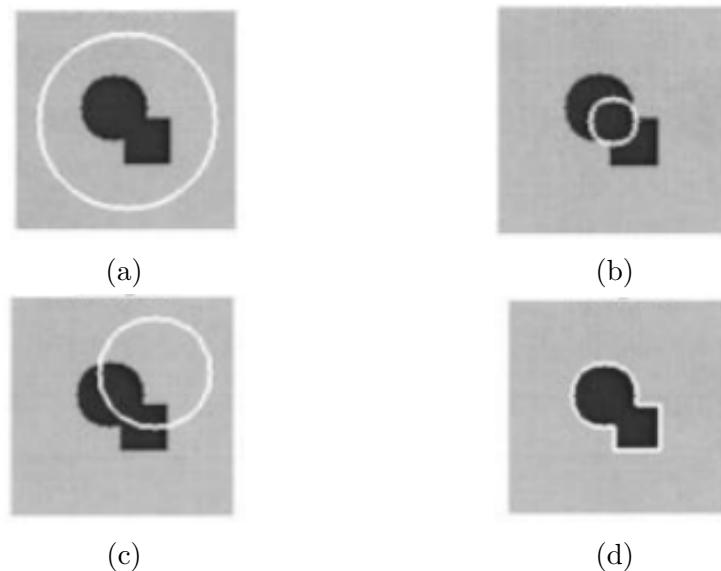


Figura 4.3: Posibles inicializaciones del contorno con el método de Chan-Vese

Fuente: [17]

Capítulo 5

Implementación de la aproximación y Ofeli

Una vez encontrada una técnica que contemplara los requisitos del proyecto, se empezó a buscar alguna implementación sobre esta técnica. En esa búsqueda se encontró un trabajo llamado Ofeli (*Open, Fast and Efficient Level set Implementation*) [3]. Este trabajo está desarrollado con Qt que es una de librerías multiplataforma para la realización de aplicaciones con GUI (*graphical user interface*) en C++. Esta aplicación es muy completa ya que, aparte del algoritmo de interés, implementa varias funcionalidades extra como varios tipos de filtrado, procesamiento, varios tipos de evolución del contorno, etc.

5.1. Ofeli

En esta sección se explicará la estructura de la implementación y las clases que se utilizan en este trabajo. Se realizará un esquema general para que el lector pueda comprender cuáles son los procesos o pautas que se dan en esta implementación para poder segmentar una imagen.

5.1.1. Estructura

La implementación está formada por varias clases que se han dividido en dos tipos: las clases o ficheros que pertenecen a la GUI (etiquetadas con la palabra «GUI-» por delante de sus nombres) y las clases que pertenecen a la implementación del algoritmo (etiquetadas con la palabra «Impl-»).

1. **Impl-ActiveContour**: clase padre de todos los tipos de contornos.
Formada por los ficheros:

- a) activecontour.cpp
 - b) activecontour.hpp
2. **Impl-ACwithoutEdges**: clase que implementa el contorno que evolucionará en una imagen a escala de grises. Formada por los ficheros:
 - a) ac_withouthedges.cpp
 - b) ac_withouthedges.hpp
 3. **Impl-ACwithoutEdgesYUV**: clase que implementa el contorno que evolucionará en una imagen a color. Formada por los ficheros:
 - a) ac_withouthedges_yuv.cpp
 - b) ac_withouthedges_yuv.hpp
 4. **Impl-list**: implementación de una lista ligada genérica. Formada por los ficheros:
 - a) linked_list.tpp
 - b) linked_list.hpp
 5. **Impl-Filters**: clase que implementa los filtros que se le pueden aplicar a la imagen antes de realizar la segmentación. Formada por los ficheros:
 - a) filters.cpp
 - b) filters.hpp
 6. **Impl-GeodesicAC**: clase que implementa un contorno geodésico. Formada por los ficheros:
 - a) geodesic_ac.cpp
 - b) geodesic_ac.hpp
 7. **Impl-HausdorffDistance**: clase que implementa la distancia de Hausdorff. Formada por los ficheros:
 - a) hausdorff_distance.cpp
 - b) hausdorff_distance.hpp
 8. **GUI-ImageViewer**: formada por los ficheros:
 - a) imageviewer.cpp
 - b) imageviewer.hpp

9. GUI-PixmapWidget: formada por los ficheros:

- a) pixmapwidget.cpp
- b) pixmapwidget.hpp

Se han resaltado las tres clases que interesarán: *ActiveContour*, *ACwithoutEdges* y *list*. Las demás clases añaden funcionalidades extra que no son necesarias en este proyecto.

5.1.2. Esquema general

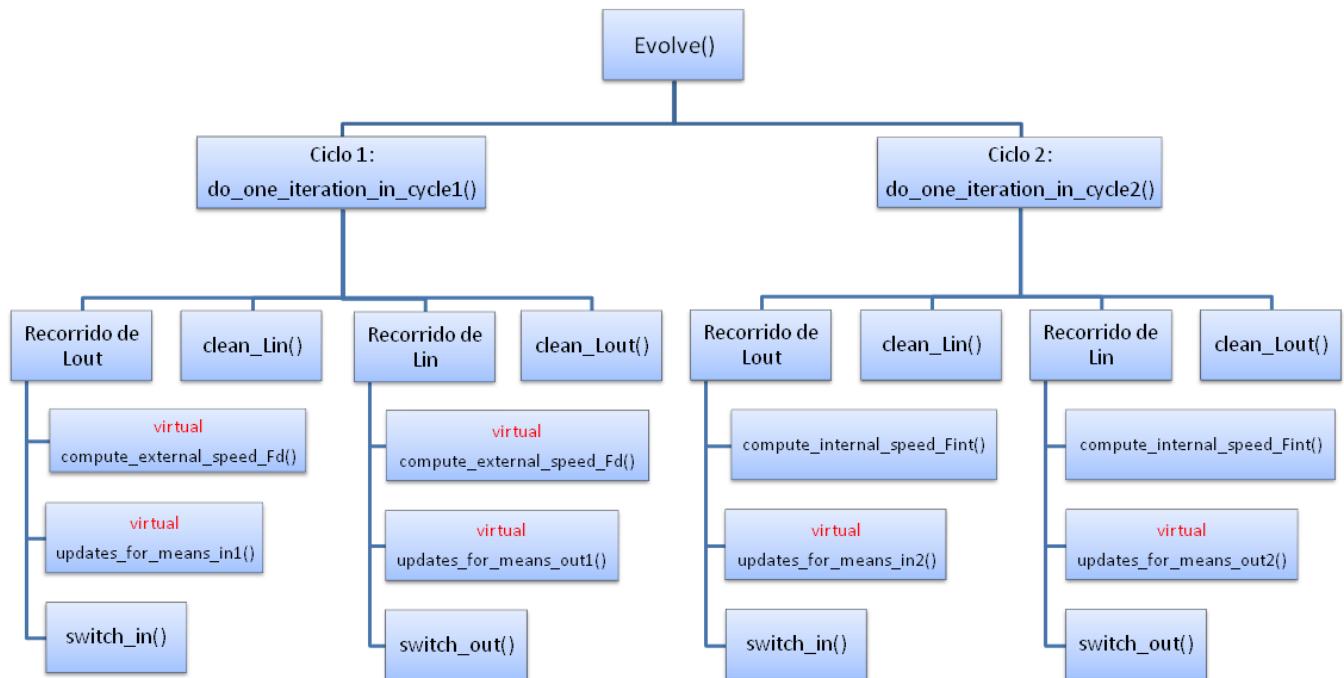


Figura 5.1: Esquema de las funciones utilizadas en el trabajo de Ofeli para realizar el algoritmo *level set* aproximado

Como se puede observar la estructura del código es prácticamente igual al algoritmo de aproximación presentado en 4.1 exceptuando que en esta implementación la velocidad de cada punto se calcula a la hora de tratar éste, no se calculan todas las velocidades de todos los puntos a la vez como se sugiere en el algoritmo 4.1. El cálculo de estas velocidades son las funciones `compute_(internal/external)_speed_(Fd/Fint)` que se pueden ver en el esquema 5.1. Las funciones `update_for_means_(in/out)(1/2)` que no están en el algoritmo de aproximación presentado en 4.1 se realizan para poder «adaptar»

el contorno a la imagen de manera que éste pueda evolucionar independientemente de los niveles de gris que se estén utilizando en la imagen. Esto se refiere a que se va haciendo una media de las intensidades de los puntos pertenecientes a las listas que representan implícitamente el contorno, es decir, L_{out} y L_{in} , de manera que el umbral con el que se va decidiendo la velocidad de cada punto va cambiando en función de los puntos que representan el contorno.

Las funciones definidas como *clean()* son las limpiezas de las listas sobre puntos redundantes que pudiera haber que se nombra en el algoritmo 4.1. Debe de quedar claro que estas funciones recorren las listas completamente al igual que la evolución de cada una de ellas.

Otra cuestión a comentar es la etiqueta «virtual» que tienen varias funciones en el esquema 5.1 que significa exactamente, que la función es virtual. Esta característica es importante a la hora de entender la jerarquía de las clases y con ello la estructura del trabajo Ofeli. Una función virtual en el lenguaje de programación C++ se utiliza en cuestiones de herencia y polimorfismo, de manera que clases hijas puedan redefinir funciones definidas como virtuales en la clase padre. Así pues, las funciones *compute_external_speed_Fd* y *update_for_means_(in/out)(1/2)* están definidas con esta cláusula ya que dependen de los datos, en este caso de las imágenes, y hay varias clases para trabajar con ellas en este trabajo. Si son imágenes a color se utilizará la clase *ACwithoutEdgesYUV*, mientras que si es una imagen a escala de grises se trabajará con la clase *ACwithoutEdges*, como se ha explicado en el anterior apartado.

Comentadas estas cuestiones se deduce que todas las funciones presentadas en el esquema 5.1 están implementadas en *ActiveContour*, la clase padre, exceptuando aquellas que tienen la etiqueta «virtual» que las implementarán las clases hijas *ACwithoutEdges* y *ACwithoutEdgesYUV* dependiendo de las características de la imagen.

Para finalizar, la implementación de la lista ligada se utiliza para representar las listas L_{out} y L_{in} con las que estaremos trabajando continuamente en los dos ciclos.

Capítulo 6

Paralelización de la aproximación del algoritmo *level set*

Las clases a modificar sobre el trabajo de Ofeli [3] para poder realizar una implementación paralela son: *ActiveContour*, *ACwithoutEdges* y *list*. De manera que se paralelizará el código para realizar la segmentación de imágenes a escala de grises, ya que en un principio el cliente sólo está interesado en este tipo de imágenes. Aún así, una vez realizada la segmentación para este tipo de imágenes sería muy sencillo poder aplicarlo a imágenes a color, ya que, como se ha visto en el capítulo anterior, la mayoría de las funciones son comunes a la clase *ActiveContour*.

La segmentación a resolver es en cierto modo «poco» costosa, ya que se realiza en unos cuantos segundos. Las máquinas disponibles para la realización de este proyecto son una máquina propia del cliente para realizar pequeñas pruebas (4 *cores*), la máquina propia del autor de esta memoria (2 *cores*) y una máquina de la Facultad de Informática de la UPV/EHU (48 *cores*). Por todo ello, se ha elegido realizar la paralelización mediante OpenMP[18] por lo que las pruebas se realizarán en máquinas con una arquitectura SMP. Además, esta elección puede suponer una ventaja a la hora de querer realizar la segmentación de las imágenes de un vídeo ya que se podría combinar con el estándar MPI como se explicará en la sección 9.1. Para aprovechar las mejores opciones que esta API(*Application Programming Interface*) nos ofrece, es decir, para poder sacarle el máximo rendimiento posible al algoritmo, se ha consultado un tutorial de *Livermore Computing Center*, uno de los centros computacionales de primera clase del mundo [19].

6.1. Planteamiento de la paralelización

Para empezar, se ha tenido que determinar donde es necesaria la paralelización. Fijándonos un poco en el código de cada ciclo, ya sea el de evolución del contorno o el del suavizado *Gaussiano*, tiene en total cuatro bucles o recorridos de las listas. Estos recorridos son los que tendremos que parallelizar ya que son los «cuellos de botella» de este algoritmo. Nótese que la cantidad de puntos que contienen las listas puede ser muy grande a medida que se aumenta el tamaño de la imagen. Además, se deberán de hacer cuatro recorridos únicamente para expandir el contorno un píxel hacia fuera o hacia dentro.

Una vez aclarada esta cuestión se debe observar más detalladamente cada recorrido de las listas en busca de condiciones de carrera. Así pues, se desglosan los cuatro recorridos comunes a los dos ciclos por separado:

1. **Primer bucle:** recorrido de la lista L_{out} . La operación que se lleva a cabo en este bucle es la denominada *switch_in()* y se realiza/ejecuta solo si se cumple cierta condición dependiente únicamente de la velocidad del punto. Como recuerdo, esta operación trasladaba el punto tratado, perteneciente a la lista L_{out} , a la lista L_{in} (al comienzo de la lista) y añadía la nueva vecindad de este punto a L_{out} (también al comienzo de la lista).
 - a) Problema:
 - 1) Cuando dos *threads* quieran cambiar el punto que están tratando de la lista L_{out} a la lista L_{in} , al estar trabajando con listas ligadas, el querer añadir al mismo tiempo dos puntos al comienzo de la lista daría un problema con los punteros. Por lo tanto, esta operación de cambiar de lista un punto deberá de hacer atómicamente.
 - 2) Al haber cambiado un punto de una lista a otra, en nuestro caso de L_{out} a L_{in} se añade la nueva vecindad de ese punto a L_{out} . Pasa exactamente lo mismo que en el anterior caso, cuando varios *threads* están añadiendo puntos vecinos al principio de la lista habrá un problema con los punteros.
 - 3) Otro problema relacionado con el anterior puede suceder cuando dos puntos de diferentes *threads* quieran añadir al mismo vecino al mismo tiempo. Esto puede suceder dependiendo de la morfología de las islas de la imagen. Véase la figura 6.1 donde se supone que los puntos de la parte superior los procesa el primer *thread* y los de la parte inferior el segundo *thread*. En

en este ejemplo el punto A sería un vecino común para el punto directamente superior a él tratado por el primer *thread* y para el punto directamente inferior a él, tratado por el segundo *thread*. Si esto ocurriese, podría haber puntos repetidos en la lista, poniendo en riesgo su consistencia.

b) Solución:

- 1) La solución más sencilla para resolver los dos primeros problemas sería poner secciones críticas a la hora de añadir puntos a las listas. Obviamente esto resuelve el problema pero el rendimiento sería prácticamente el de la ejecución en serie si no es incluso peor que éste por el *overhead* que pueda suponer la gestión de la sección crítica. Entonces, si el problema son las operaciones con las listas, está claro que cada *thread* tendrá que tener una estructura propia donde añadir esos puntos para poder evitar estos dos primeros problemas. Así pues, se ha decidido «partir» las listas L_{out} y L_{in} en función del numero de *threads*, para que cada uno trabaje con sus listas. Al final del ciclo habrá que volver a unir todos los «trozos» que tenga cada *thread* para volver a reconstruir las listas. Estas operaciones pueden suponer un sobrecoste alto en cuanto a la versión serie, aún así, valiéndonos de las posibilidades que nos ofrecen las listas enlazadas podemos llegar a hacer estas dos operaciones casi en tiempo constantes. En el caso de no poder hacerlas eficientemente se podrían realizar cada cierto número de ciclos, aunque esto tendrá como consecuencia que la carga esté desbalanceada entre distintos *threads*, ya que cada uno puede quitar o añadir una diferente cantidad de píxeles. Habrá que buscar un equilibrio para poder hacer un buen *load balancing*.
- 2) En cuanto al tercer problema, no queda más opción que resolverlo mediante una sección crítica, en la que se asegure la inserción única del vecino.

2. **Segundo bucle:** limpieza de la lista L_{in} . En este recorrido se comprueba si hay algún punto redundante en base a su vecindad y si alguno lo es se quita de la lista.

a) Problema:

- 1) Al realizar el borrado de un punto de la lista, y al trabajar con listas enlazadas, puede que otro *thread* este tratando el

elemento posterior al que se vaya a eliminar y se tenga problema con los punteros de nuevo. Hay que decir también que esta situación solo se podrá dar con los puntos «frontera», es decir, con el último punto de un *thread* y el primero de otro.

b) Solución:

- 1) Este problema quedaría resuelto con la solución propuesta para los primeros problemas del primer bucle, es decir, el troceado de las listas en base al número de *thread*: un trozo por cada uno de ellos.
3. **Tercer bucle:** recorrido de L_{in} . Los problemas y soluciones de este bucle son exactamente iguales a los del primero ya que son simétricos, es decir, que en este caso las operaciones serán las inversas. El cambio de un punto que cumple con la condición de velocidad esta vez será de L_{in} a L_{out} y se añadirán los puntos de éste a L_{in} .
4. **Cuarto bucle:** limpieza de L_{out} . De la misma manera que el tercer bucle los problemas y soluciones del segundo bucle serán válidos para éste también al ser el caso contrario, la limpieza de L_{out} .

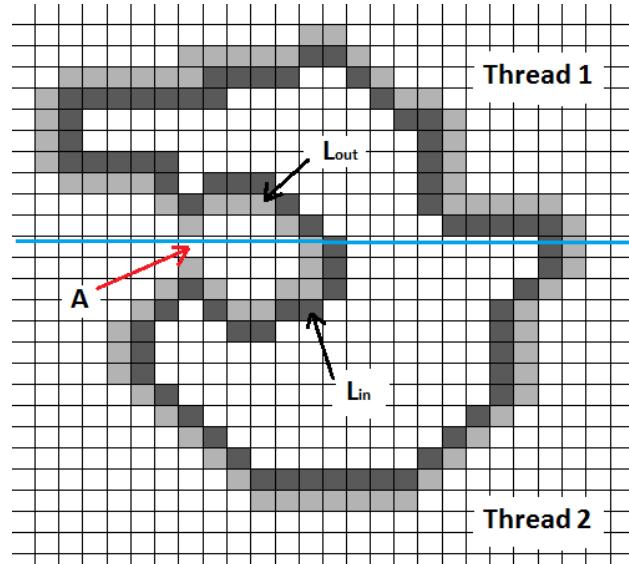


Figura 6.1: Ejemplo de condición de carrera al querer añadir un punto vecino en común a puntos tratados por distintos *threads*

6.1.1. Primera implementación

En esta sección se explican las características de la primera implementación paralela realizada siguiendo el análisis del anterior apartado. Así pues, cada ciclo quedará estructurado como se ve en la figura 6.2. Las nuevas características son de color rojo y se han marcado con un asterisco (*).

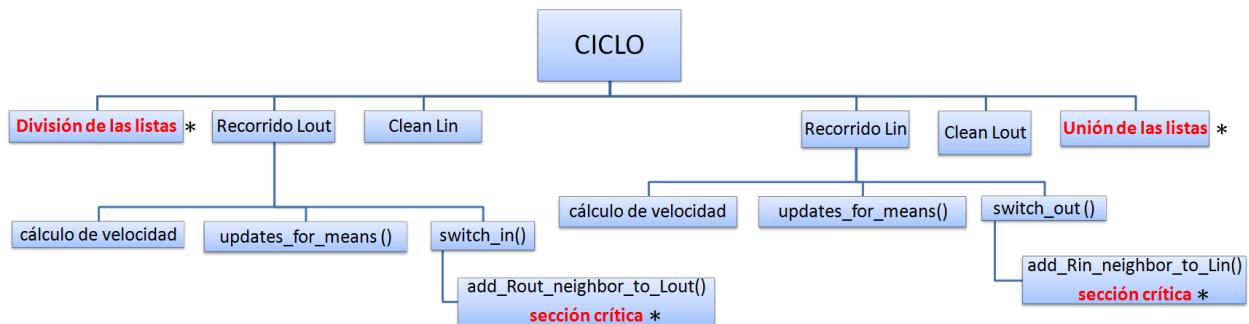


Figura 6.2: Esquema de la primera implementación

En esta primera implementación se invirtió mucho tiempo ya que los cambios realizados fueron muchos. Se irá paso a paso comentando las características de cada cambio, así como la razón por la que se llevo a cabo dicho cambio.

Lista enlazada

Muchos han sido los cambios realizados en comparación con la implementación de lista enlazada creada en el trabajo Ofeli. Todos los cambios aquí realizados han tenido como objetivo minimizar el coste operacional de las operaciones necesarias para realizar la paralelización.

1. Se ha insertado en cada nodo un puntero hacia el elemento anterior, es decir, se ha creado una lista doblemente ligada.
2. La modificación anterior ha dado pie a crear un apuntador al final de la lista, un *tail*.
3. Se ha convertido el tiempo de la ejecución de la función size() en orden constante en lugar de la implementación computacionalmente lineal (al número de elementos de la lista) que estaba.
4. Otras modificaciones que no influyen en la implementación paralela.

División de la lista

Las listas L_{out} y L_{in} se dividen en un número de trozos o sublistas igual al número de *threads* de manera que cada uno tenga la parte de la lista original con la que trabajará.

Al inicio del programa, después de inicializar las dos listas con el contorno definido manualmente, se crean tantos punteros o apuntadores como *threads* vayan a trabajar en la ejecución. Cada puntero se coloca apuntando al primer elemento que tratará cada *thread*, es decir, suponiendo que tenemos 80 elementos a repartir entre cuatro *threads*, el primer puntero apuntará al primer elemento, ya que será el primer elemento a tratar por el primer *thread*, el segundo puntero apuntará al vigésimo primer elemento y así sucesivamente. Véase la figura 6.3 para ver un ejemplo de ello. Esta primera recolocación de los *heads* de cada *thread* se realiza en tiempo lineal al número de elementos en las listas, sin embargo, esta operación sólo se realiza en la preparación de la evolución y por lo tanto, una sola vez.

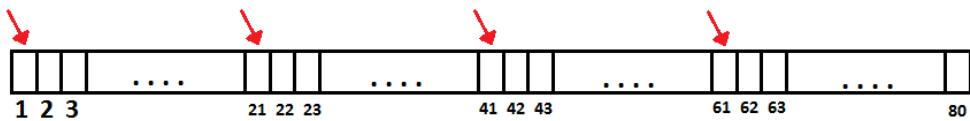


Figura 6.3: Esquema de la colocación de los *heads* de trozo de lista a crear

Por lo tanto, se accede en un orden constante a cada elemento *head* creado, se establecen los *tails* de las sublistas accediendo al anterior elemento de estos *heads*, se establecen los tamaños de cada sublista (posible conociendo la posición de cada *tail* y llevando una cuenta) y se «rompen» los enlaces con sus anteriores elementos de manera que se separen por completo las sublistas. Todo ello es posible gracias a las modificaciones realizadas en la implementación de la lista ligada. Véase la figura 6.4 para ver un ejemplo de todo ello.

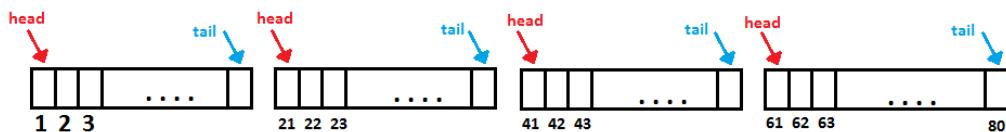


Figura 6.4: Ejemplo de creación de cada sublista usando los *heads* establecidos

Unión de las sublistas

Siguiendo el procedimiento inverso a la división de las listas se realiza la unión. En este caso, sabiendo los *heads* y los *tails* de cada sublista, se vuelve a crear la unión entre ellas, más concretamente, entre el *tail* y el *head* de la siguiente lista. El tamaño de la lista completa será la suma de los tamaños de todas las sublistas. Véase la figura 6.5 como explicación de ello.

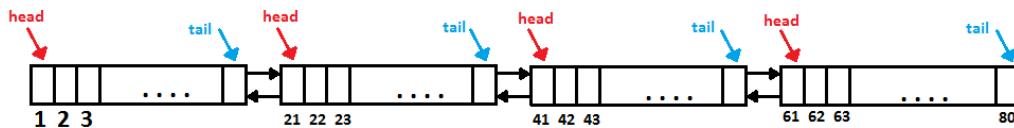


Figura 6.5: Ejemplo de unión de cada sublista usando los *heads* y *tails* de cada una

Al realizar esta unión, también se aprovecha a reinicializar los *heads* que se utilizarán en la siguiente división de la lista. Los *heads* se inicializarán previamente como los *heads* de cada sublista. Nótese que estos *heads* se habrán posicionado muy cerca de la posición óptima, ya que habitualmente todas las zonas del contorno tienden a expandirse por igual. No obstante, se realiza un centrado de cada *head*, es decir, se mueve el puntero hasta la posición exacta en la que debe estar, si es que no lo está, realizando la división del número de puntos entre el número de *thread* de la ejecución. Los *tails* no harán falta establecerlos ya que se posicionan en la división de las listas. Véase la figura 6.6 como explicación de ello. El número de operaciones a realizar comparadas con el número de puntos en la lista es pequeño.

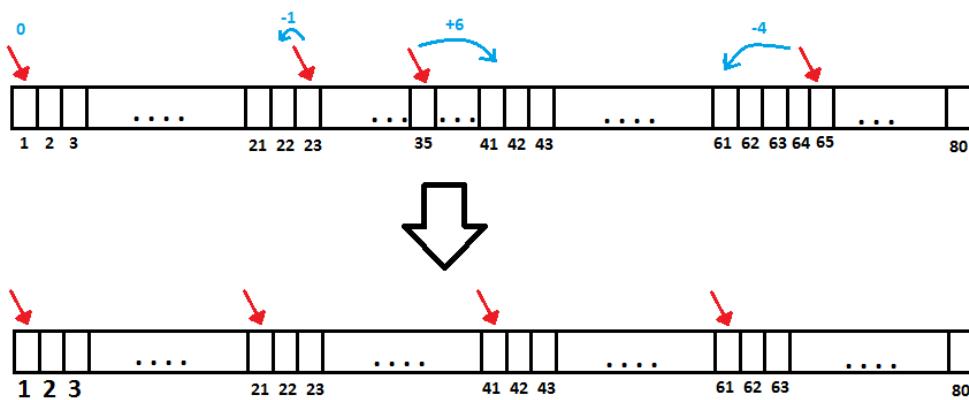


Figura 6.6: Ejemplo de unión de cada sublista usando los *heads* y *tails* de cada una

Secciones críticas

Las secciones críticas a realizar para resolver la condición de carrera se han establecido en las funciones de añadir un vecino del punto que se cambia de una lista a otra. Se ha seguido un esquema *Test and Test-and-set* para minimizar la contención de la sección crítica ya que se realiza antes la condición que se debe de hacer en la sección crítica antes de entrar a ésta.

A continuación se presenta el código de la sección crítica.

Código

```

void ActiveContour::add_Rout_neighbor_to_Lout(int
    neighbor_offset, int tid)
{
    //Test, test and set
    if( phi[neighbor_offset] == 3 ) // exterior value
    {
        #pragma omp critical
        {
            if( phi[neighbor_offset] == 3 ) // exterior value
            {
                phi[neighbor_offset] = 1; // outside boundary
                value

                Splited_Lout[tid]->push_front(neighbor_offset);
            }
        }
    }
    return; }

void ActiveContour::add_Rin_neighbor_to_Lin(int
    neighbor_offset, int tid)
{
    //Test, test and set
    if( phi[neighbor_offset] == -3 ) // interior value
    {
        #pragma omp critical
        {
            if( phi[neighbor_offset] == -3 ) // interior value
            {
                phi[neighbor_offset] = -1; // inside boundary
                value
            }
        }
    }
    return; }

```

```

        Splited_Lin[tid]->push_front (neighbor_offset) ;
    }
}
}

return;
```

6.1.2. Rendimiento

Realizada la primera implementación, llega la hora de comprobar su rendimiento y ver la mejora obtenida respecto a la versión serie. En estas primeras pruebas se ha utilizado únicamente una imagen de tamaño 3000x2800 y que es exactamente la que se muestra en la figura 1.1d.

Las pruebas realizadas han sido en un ordenador de la Universidad del País Vasco. Este ordenador tiene una arquitectura SMP, con una memoria principal de 64GB y dispone de cuatro procesadores *AMD Opteron(tm) Processor 6168* los cuales tienen doce *cores* que trabajan a 1900 MHz. Por lo tanto, se disponen de 48 *cores* para poder realizar las pruebas. También se ha considerado añadir que se ha tenido que separar la parte gráfica del trabajo original de Ofeli y extraer el algoritmo de evolución para poder realizar las pruebas.

La compilación se ha realizado con el compilador gcc (versión 4.6.3) con la siguiente línea de comandos:

```
g++ main.cpp activecontour.cpp ac_withoudedges.cpp -o prueba
-O3 -fopenmp -Dcimg_use_png -lpng -lz -Dcimg_display=0
```

Mientras no se diga lo contrario, todas las ejecuciones se realizarán en la misma máquina y los tiempos de todas las tablas que aparezcan serán la media de cinco ejecuciones. A esta primera implementación se le nombrará como «sección critica normal».

Número de threads	Serie	2	4	8	16	32	48
Tiempo (s)	16,52	11,63	7,80	9,38	14,37	27,66	31,37

Speed-up	1,00	1,42	2,12	1,76	1,15	0,60	0,53
Eficiencia	100,00 %	71,03 %	52,98 %	22,02 %	7,19 %	1,87 %	1,10 %

Tabla 6.1: Rendimiento de las ejecuciones de la primera implementación paralela

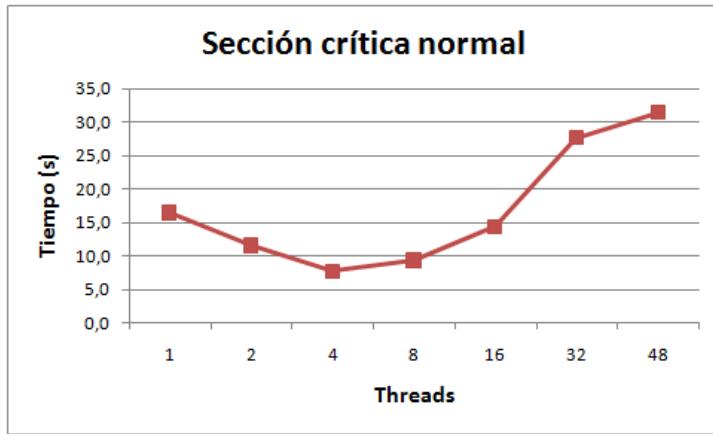


Figura 6.7: Gráfica de los tiempos de ejecución de la primera implementación paralela

Como se puede observar en la figura 6.7, esta primera implementación sólo escala bien hasta los cuatro *cores*.

Conclusión

Se analizan los tiempos de ejecución y se comprueba que, efectivamente, el cuello de botella está en el acceso a la sección crítica. Véase el desglose del tiempo de ejecución (2 *cores*) de un ciclo en la tabla 6.2. Aquí se puede observar que la mayoría del tiempo se pierde en realizar los bucles 1 y 2, es decir, donde están las funciones con la secciones críticas. Esto sucede ya que ésta se hace prácticamente siempre, debido a que el contorno suele tender a expandirse, y puede que se repita varias veces por cada punto, al realizarse por cada nuevo vecino. Por otro lado, la ejecución es únicamente con dos *cores*, por lo que la contención de la sección crítica será mayor con más *cores*, razón por la que esta primera implementación no escala bien.

	División listas	1º bucle	Clean Lin	2º bucle	Clean Lout	Unión listas	POR CICLO
Serie	0,0	0,0127	0,008	0,0145	0,0071	0,0	0,043
Óptimo 2 threads	0,0	0,0063	0,0040	0,0072	0,0035	0,0	0,0215
Paralelo 2 threads	0,0	0,0093	0,0041	0,0109	0,0038	0,0	0,0281
Sobrecoste	0,0 %	46,0 %	2,1 %	51,3 %	8,6 %	0,0 %	

Tabla 6.2: Desglose de tiempos (s) de un ciclo del algoritmo con una ejecución paralela con dos *cores* de la primera implementación

6.2. Primer paso de mejora: segunda implementación

Concluido ya que el problema está en la sección crítica se intenta reducir las operaciones dentro de ésta reduciendo así el tiempo que cada *thread* está dentro de ella. La «sección crítica mejorada» tendrá únicamente dos asignaciones, reduciendo considerablemente el tiempo dentro de ella. Se utilizará un *flag* para saber qué *thread* ha sido el que ha entrado a la sección crítica para posteriormente hacer la operación correspondiente. En este caso obviaremos la segunda función ya que tiene las mismas características que la primera.

Código

```
void ActiveContour::add_Rout_neighbor_to_Lout(int
neighbor_offset,int tid)
{
    bool flag = false;

    if( phi[neighbor_offset] == 3 ) // exterior value
    {
        #pragma omp critical
        {
            if( phi[neighbor_offset] == 3 ) // exterior value
            {
                phi[neighbor_offset] = 1; // outside boundary
                value
                flag = true;
            }
        }
        if(flag)
            Splited_Lout[tid]->push_front(neighbor_offset);
    }
    return;
}
```

6.2.1. Rendimiento

Núm. <i>threads</i>	Serie	2	4	8	16	32	48
Tiempo (s)		16,52	10,74	6,58	5,42	3,68	6,02
Speed-up		1,00	1,54	2,51	3,05	4,49	2,74
Eficiencia		100,0 %	76,9 %	62,8 %	38,1 %	28,1 %	8,6 %
							5,1 %

Tabla 6.3: Rendimiento obtenido de las ejecuciones de la segunda implementación paralela

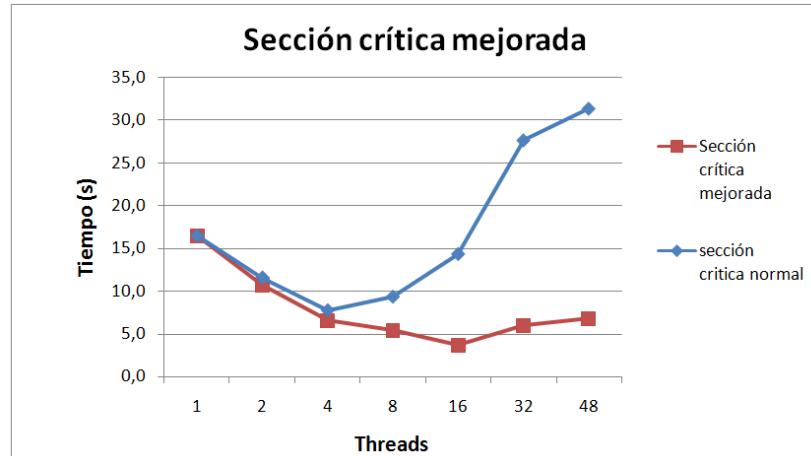


Figura 6.8: Gráfica de los tiempos de ejecución de la primera y segunda implementación

Como se puede ver en el gráfico 6.8 la mejora es bastante notable. Esto se debe a que se ha reducido notablemente la contención entre los *threads* al no tener prácticamente operaciones a realizar dentro de la sección crítica. El tiempo de ejecución del sistema se reduce hasta los 16 cores.

Conclusión

A pesar de haber mejorado mucho los tiempos el resultado no sigue siendo óptimo del todo ya que la implementación escala poco a poco hasta los 16 *threads* y luego el tiempo va aumentado. Hay que intentar reducir aún más la contención de alguna manera. Analizando un poco más a detalle, la sección

crítica se realiza para cualquier vecino que se quiera añadir después de haber expandido o contraído un punto. Es decir, que no importa la localización de los puntos que se estén tratando, habrá colisión en la sección crítica. En realidad, la sección crítica se puso para evitar que se pudieran añadir varios puntos vecinos a la vez. ¿Hay alguna manera de realizar secciones críticas dependientes del punto? Sería la pregunta clave a responder para poder resolver el problema.

6.3. Segundo paso de mejora: tercera implementación

Buscando alguna manera de realizar secciones críticas dependiente del punto se deciden usar semáforos o *locks* en inglés. Se podría tener una matriz de *locks* de la misma dimensión que la imagen y cada vez que se vaya a trabajar con un punto cerrar el *lock* que le pertenece. El problema de esta idea de tener un *lock* para cada píxel, en una imagen de 3000x2800, sería la cantidad de memoria y la complejidad de gestión de éstos, ya que el número de *locks* necesarios sería muy elevado. Como alternativa y buscando una solución de compromiso se ha implementado un esquema de *locks* por zonas. En vez de tener un *lock* para cada punto se podría tener un número de *locks* no muy grande para bloquear cierta zona de la imagen.

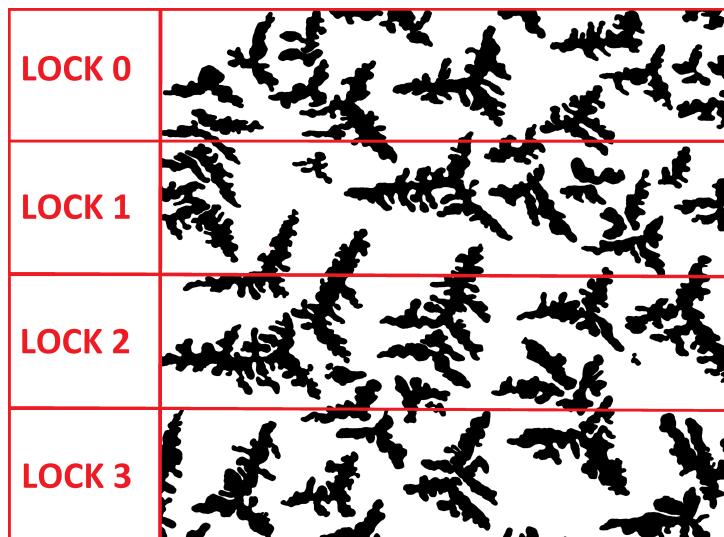


Figura 6.9: Ejemplo de una malla de cuatro locks de establecidos en forma rectangular

En la imagen 6.9 se puede observar un ejemplo de una malla de *locks* en la número de *locks* establecido es 4 y tiene una forma rectangular. El número de óptimo de locks a establecer será lo que se trabaje en esta tercera implementación. Además, el lector podría pensar que se pueden realizar distintas figuras geométricas en la creación de la estructura de *locks*, aunque esta opción no se realizará en este proyecto ya que la estrategia de distribución de los puntos en las sublistas se realiza recorriendo la matriz por filas, lo que significa que los rectángulos darán una mejor respuesta temporal ya que habrá poca colisión entre los *threads*.

Código

```

void ActiveContour::add_Rout_neighbor_to_Lout (int
    neighbor_offset, int tid)
{
    bool flag = false;

    if( phi[neighbor_offset] == 3 ) // exterior value
    {
        int lockNumber = ( (neighbor_offset/img_width)
            *numLocks)/img_height;

        omp_set_lock(&(locks[lockNumber]));

        if( phi[neighbor_offset] == 3 ) // exterior value
        {
            phi[neighbor_offset] = 1; // outside boundary value
            flag = true;
        }

        omp_unset_lock(&(locks[lockNumber]));

        if(flag)
            Splited_Lout[tid]->push_front(neighbor_offset);
    }
    return;
}

```

6.3.1. Rendimiento

THREADS	Locks						su	Efi.
	4	8	16	24	32	50		
1	16,69	16,64	16,63	16,61	16,79	16,73	-	-
2	10,49	10,64	10,35	10,52	10,35	10,33	1,59	79,6 %
4	6,79	7,03	6,86	6,85	6,73	6,60	2,46	61,5 %
8	4,22	5,73	5,72	5,42	5,44	5,40	3,96	49,5 %
16	3,99	3,93	4,89	3,26	4,87	4,84	4,18	26,1 %
20	3,29	3,87	3,88	3,19	3,89	3,89	5,08	25,4 %
24	2,95	3,28	3,29	3,32	3,30	3,29	5,65	23,5 %
28	3,03	2,96	2,93	2,92	2,96	2,93	5,52	19,7 %
32	2,85	2,72	2,80	2,70	3,00	2,80	5,86	18,3 %
36	3,74	3,81	3,29	3,48	2,85	3,28	4,46	12,4 %
40	4,12	4,06	3,79	3,98	3,85	3,81	4,05	10,1 %
44	4,00	4,06	4,05	3,83	3,75	3,74	4,18	9,5 %
48	3,91	3,91	3,77	3,68	3,82	3,66	4,27	8,9 %

Tabla 6.4: Tiempos de ejecución (s), eficiencia y *speed-ups* (su) obtenidos con distintas combinaciones de *threads* y *locks* en la tercera implementación paralela

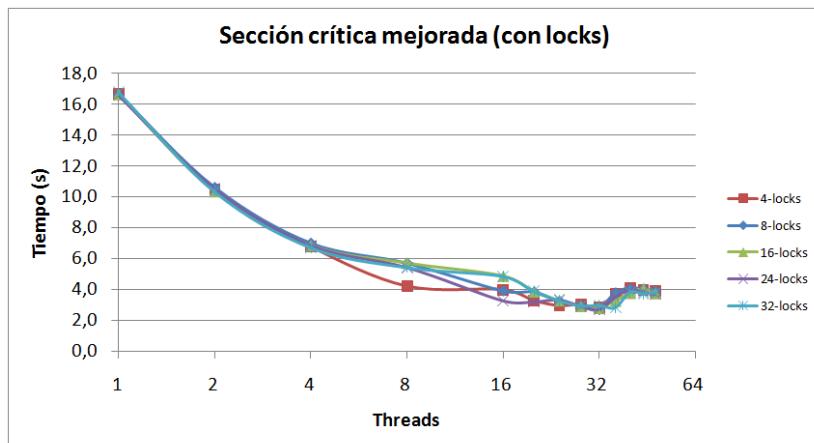


Figura 6.10: Gráfico de los resultados obtenidos en este tercera implementación

Como se puede observar en la figura 6.10 se ha mejorado aún más la respuesta temporal, escala bien hasta los 32 *threads* en vez de hasta los 16

que se conseguían con la anterior implementación (véase la figura 6.8). Las ejecuciones más rápidas se logran con la combinación de 32 *threads* y con 24 *locks*, teniendo un resultado de 2,7 segundos.

Por otro lado, también se han realizado pruebas con la primera implementación, es decir, con la «sección crítica normal» para analizar el efecto de la reducción de la contención mediante la solución de *locks* distribuidos en dicha implementación. Esto se refiere a que la sección la crearán los *locks* pero lo que hay dentro será lo que había en la implementación de la «sección crítica normal». Los resultados son los que se muestran en el gráfico 6.11. Como se puede observar el tiempo obtenido es prácticamente el conseguido con la última implementación realizada. Esto quiere decir que el problema de todo ello era la contención que había con todos los *threads* y no importa tanto el tiempo que se pasa dentro de la sección crítica. Sin embargo, también ahí que decir que con la implementación de la «sección crítica normal» son necesarios más *locks* para evitar esa contención entre los *threads* ya que la sección crítica es más larga.

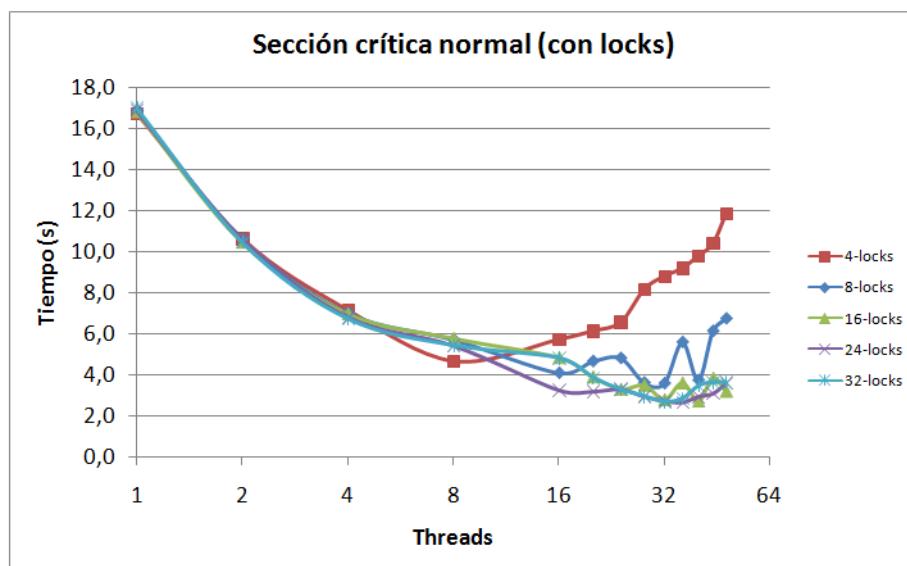


Figura 6.11: Gráfica de los tiempos de ejecución con los *locks* y la estructura de la «sección crítica normal»

Conclusión

Ya que los resultados obtenidos con la última implementación y la primera son parecidos no se podrá reducir mucho más el tiempo en cuanto a la contención, ya que parece que a partir de cierto número de *locks* ésta

desaparece. Así pues, la reducción tendría que hacerse por otra vía, o bien intentar reducir el tiempo que se realiza en paralelo o entrar en más detalle como la compartición falsa en la memoria caché que pudiera haber con ciertas variables de la implementación. El propio tiempo de gestión de los *locks*, la compartición falsa en caché que pudiera haber con ciertas variables, la gestión de la sección paralela y la división y unión de las listas parece que añaden en total un *overhead* que se ha decidido dejar fuera del ámbito de este proyecto.

Capítulo 7

Experimentación

En este capítulo mostraremos el rendimiento de la mejor versión implementada, la tercera versión, aquella con la «sección critica mejorada» y haciendo uso de la malla de *locks* con mas detalle. En las anteriores presentaciones de los rendimientos de las distintas implementaciones se ha utilizado un tipo de imagen, tamaño de esta e inicialización inicial del contorno. En este apartado realizaremos pruebas más completas, combinando distintos factores para ver mas detalladamente el comportamiento de la mejor versión de la paralelización. Además, se podrá llegar a concluir que tipo de combinación beneficia a la evolución del contorno en cierto tipo de imágenes.

Se ha decidido probar 3 tipos de imágenes diferentes, con varios tamaños de estas, varios tamaños de las mallas de *locks* y varios *threads*. En la figura 7.1 se puede observar los tres tipos de imágenes diferentes que se utilizarán en la realización de dichas pruebas.

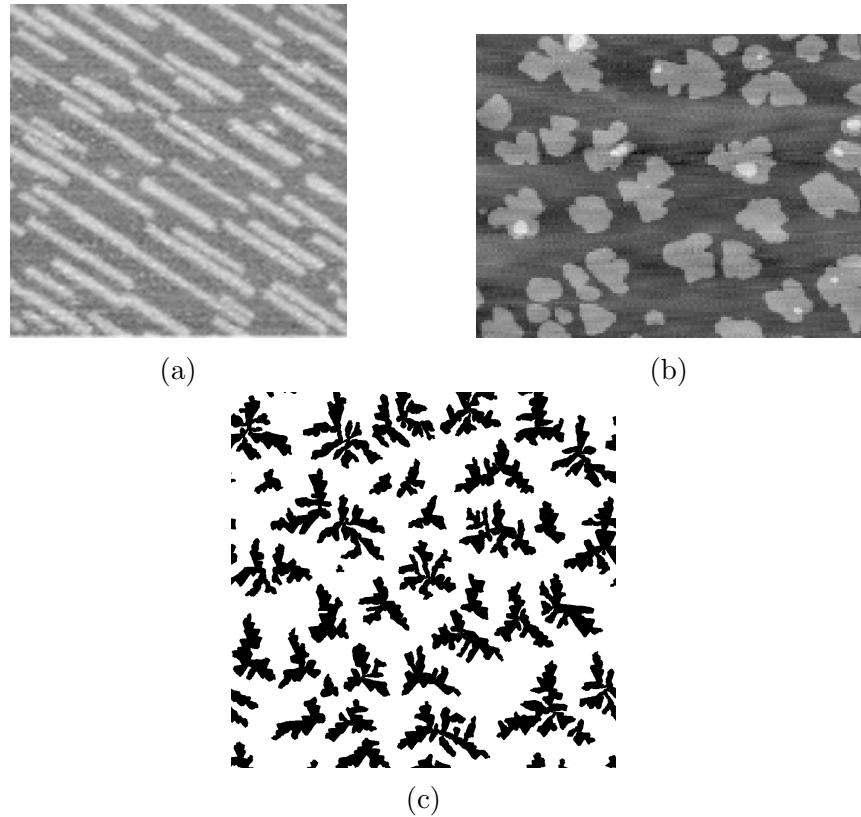


Figura 7.1: Tipos de imágenes con las que se realizarán las pruebas completas de la tercera implementación

7.1. Primera imagen

La primera imagen a probar será la imagen 7.1a y a continuación se presentan los resultados de las ejecuciones con diferentes tamaños de la imagen.

500x500

Threads \ Locks	4	8	16	24	32	50
1	2,01	2,03	2,03	2,04	2,04	2,03
2	1,34	1,37	1,37	1,34	1,35	1,35
4	1,00	0,96	0,96	0,97	0,96	0,95
8	0,62	0,91	0,91	0,80	0,88	0,90
16	0,62	0,56	0,56	0,48	0,74	0,70
20	0,53	0,64	0,64	0,50	0,67	0,60
24	0,43	0,52	0,52	0,56	0,56	0,49
28	0,47	0,43	0,43	0,45	0,43	0,47
32	0,42	0,40	0,40	0,39	0,38	0,39
36	0,44	0,37	0,37	0,35	0,34	0,36
40	0,44	0,32	0,32	0,31	0,32	0,32
44	0,45	0,35	0,35	0,31	0,31	0,30
48	0,45	0,36	0,36	0,33	0,34	0,37

Tabla 7.1: Tiempos de ejecución de la imagen 7.1a con un tamaño de 500x500

Como se puede observar en la tabla 7.1 los mejores tiempos se consiguen con 40 y 44 *threads* y a partir de los 24 *locks*. El menor tiempo que se consigue es de 0,3 segundos, con un *speed-up* de 6,7 y una eficiencia del 15,1 %.

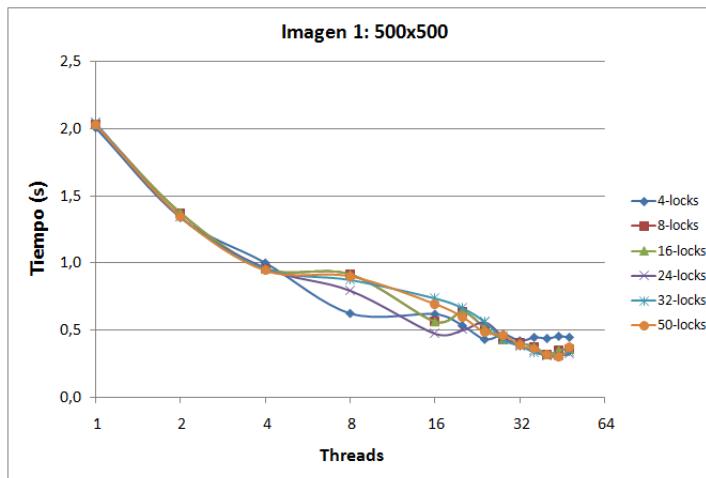


Figura 7.2: Gráfica de los tiempos de ejecución de la imagen 7.1a con un tamaño de 500x500

1500x1500

Threads \ Locks	4	8	16	24	32	50
1	14,88	14,91	14,92	14,87	14,92	14,90
2	8,52	8,54	8,42	8,54	8,52	8,50
4	5,96	5,96	5,82	5,88	5,88	5,88
8	3,34	4,89	4,87	4,80	4,80	4,80
16	3,45	3,40	4,27	2,58	4,28	4,27
20	2,86	3,50	3,51	2,72	3,43	3,43
24	2,34	2,89	2,90	2,87	2,86	2,86
28	2,63	2,50	2,52	2,50	2,47	2,49
32	2,65	2,28	2,23	2,27	2,23	2,26
36	2,64	2,10	2,20	2,19	2,21	2,23
40	2,69	2,16	2,17	2,17	2,16	2,16
44	2,66	2,05	2,07	2,08	2,06	2,05
48	2,79	2,14	2,05	2,05	2,07	2,02

Tabla 7.2: Tiempos de ejecución de la imagen 7.1a con un tamaño de 1500x1500

En esta ocasión el mejor resultado se consigue con 48 *threads* y 50 *locks*. El resultado obtenido son 2,02 segundos, un *speed-up* de 7,4 y una eficiencia del 15,4 %.

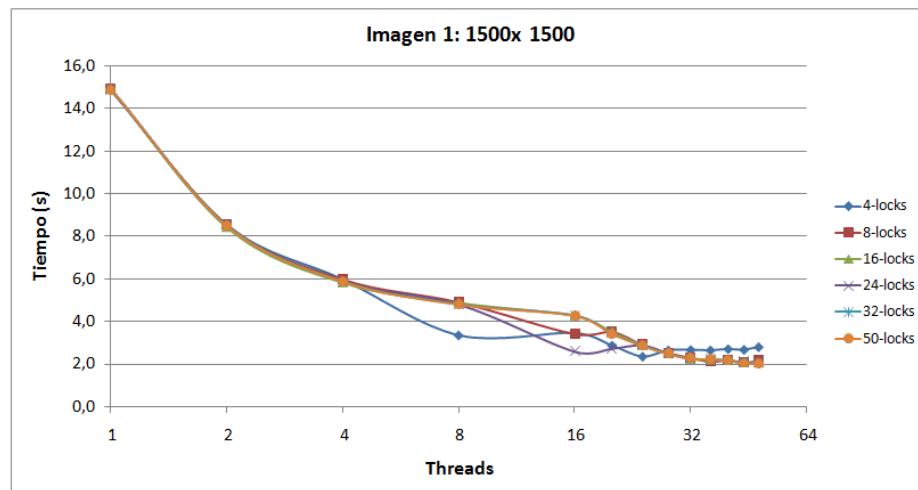


Figura 7.3: Gráfica de los tiempos de ejecución de la imagen 7.1a con un tamaño de 1500x1500

3000x3000

	4	8	16	24	32	50
1	43,38	43,28	43,32	43,34	43,24	43,19
2	28,46	26,85	26,83	26,89	26,75	26,71
4	16,94	17,25	17,18	17,00	16,99	16,80
8	9,53	13,34	13,42	13,42	13,34	13,38
16	9,59	9,50	12,01	7,36	11,98	12,00
20	7,85	9,69	9,71	7,77	9,67	9,76
24	6,81	8,30	8,29	8,25	8,20	8,24
28	7,42	7,19	7,22	7,18	7,13	7,19
32	7,55	7,06	7,03	7,04	6,98	6,83
36	7,75	6,90	7,10	7,22	7,19	6,97
40	8,00	7,05	6,93	6,91	6,90	6,88
44	8,17	6,62	6,64	6,63	6,78	6,72
48	8,02	6,64	6,52	6,49	6,75	6,58

Tabla 7.3: Tiempos de ejecución de la imagen 7.1a con un tamaño de 3000x3000

El mejor resultado es de 6,49 y se alcanza con 48 *threads* y 24 *locks*. Se obtiene un *speed-up* de 6,7 y una eficiencia del 13,9 %.

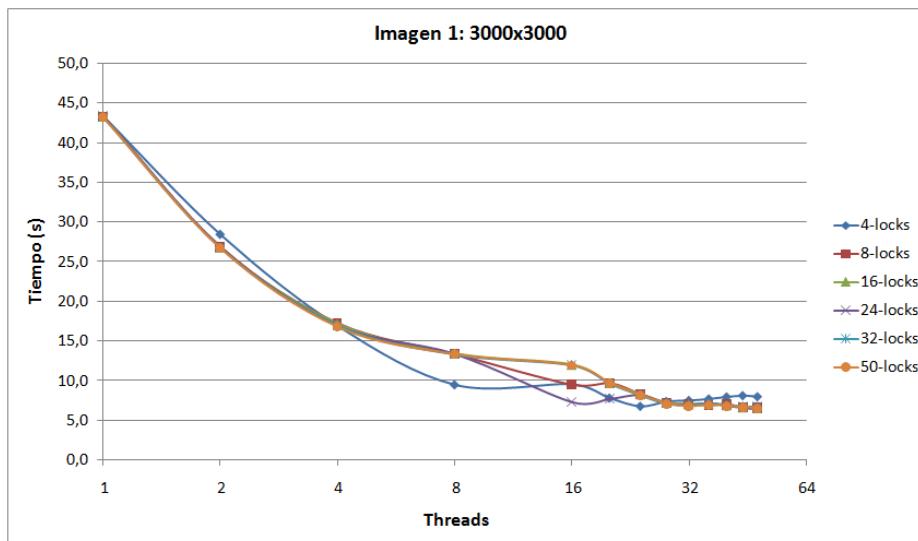


Figura 7.4: Gráfica de los tiempos de ejecución de la imagen 7.1a con un tamaño de 3000x3000

Resultado

Los resultados de la segmentación de esta primera imagen han sido los que se muestran en 7.5, donde se ha marcado la lista L_{out} de color rojo y la lista L_{in} de azul. Se ha conseguido obtener el contorno las islas que es lo que se esperaba.

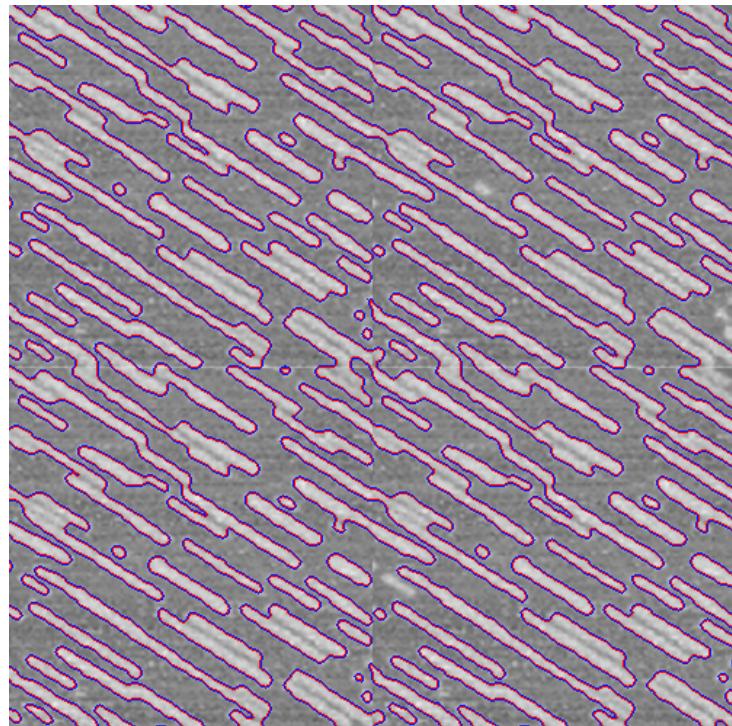


Figura 7.5: Resultado de la segmentación de la imagen 7.1a

7.2. Segunda imagen

La segunda imagen a probar será la imagen 7.1b y a continuación se presentan los resultados de las ejecuciones con diferentes tamaños de la imagen.

500x500

Threads \ Locks	4	8	16	24	32	50
1	0,51	0,51	0,52	0,51	0,51	0,52
2	0,35	0,35	0,35	0,35	0,34	0,34
4	0,25	0,25	0,25	0,25	0,24	0,24
8	0,17	0,21	0,21	0,20	0,20	0,20
16	0,16	0,15	0,18	0,13	0,18	0,18
20	0,13	0,15	0,15	0,13	0,15	0,15
24	0,13	0,14	0,13	0,13	0,13	0,13
28	0,16	0,12	0,12	0,12	0,12	0,12
32	0,17	0,11	0,11	0,11	0,11	0,11
36	0,18	0,12	0,11	0,11	0,11	0,11
40	0,20	0,11	0,11	0,10	0,10	0,11
44	0,21	0,13	0,11	0,11	0,10	0,10
48	0,23	0,18	0,14	0,13	0,14	0,15

Tabla 7.4: Tiempos de ejecución de la imagen 7.1b con un tamaño de 500x500

Como se puede observar en la tabla 7.4 los mejores tiempos se consiguen con 40 y 44 threads y a partir de los 24 locks. El menor tiempo que se consigue es de 0,10 segundos, con un speed-up de 4,9 y una eficiencia del 12,5 %.

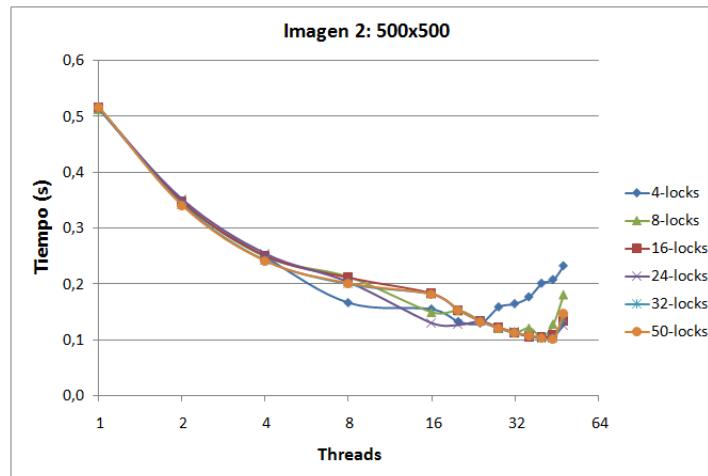


Figura 7.6: Gráfica de los tiempos de ejecución de la imagen 7.1b con un tamaño de 500x500

1500x1500

Threads \ Locks	4	8	16	24	32	50
1	5,45	5,45	5,46	5,47	5,46	5,46
2	3,56	3,61	3,57	3,56	3,54	3,54
4	2,63	2,67	2,65	2,58	2,58	2,56
8	1,68	2,23	2,20	2,11	2,11	2,10
16	1,55	1,50	1,82	1,22	1,22	1,83
20	1,31	1,53	1,53	1,21	1,23	1,53
24	1,15	1,31	1,31	1,30	1,31	1,32
28	1,43	1,17	1,19	1,18	1,19	1,17
32	1,59	1,12	1,12	1,12	1,13	1,10
36	1,82	1,27	1,26	1,24	1,27	1,31
40	1,97	1,37	1,39	1,40	1,37	1,35
44	2,03	1,40	1,38	1,41	1,36	1,39
48	2,16	1,80	1,41	1,37	1,42	1,42

Tabla 7.5: Tiempos de ejecución de la imagen 7.1b con un tamaño de 1500x1500

Como se puede observar en la tabla 7.5 el mejor resultado obtenido es de 1,10 segundos, con la combinación de 32 threads y 50 locks. Se consigue por lo tanto un *speed-up* de 4,9 y una eficiencia del 15,5 %.

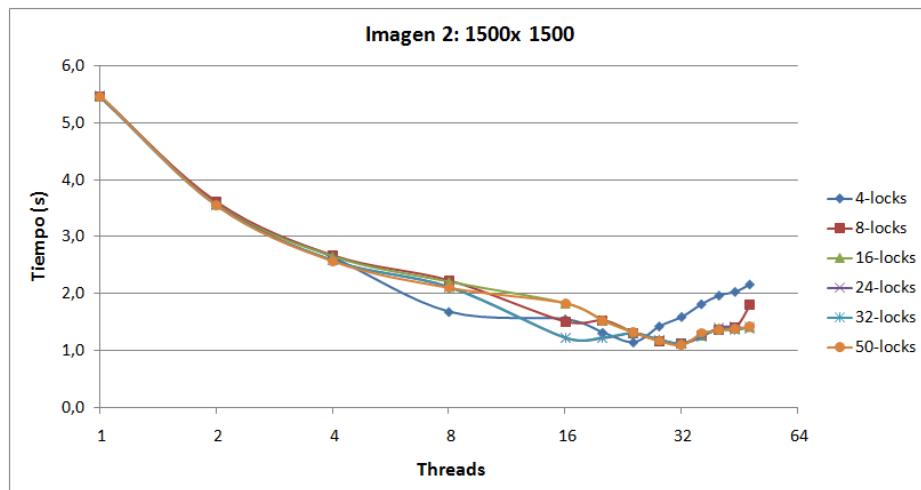


Figura 7.7: Gráfica de los tiempos de ejecución de la imagen 7.1b con un tamaño de 1500x1500

3000x3000

Threads \ Locks	4	8	16	24	32	50
1	22,29	22,40	22,25	22,29	22,27	22,31
2	9,64	9,65	9,83	9,61	9,58	9,58
4	12,42	12,41	12,64	12,20	12,21	12,06
8	5,88	7,57	7,61	7,65	7,53	7,54
16	5,67	5,43	6,74	4,48	6,71	6,72
20	4,62	5,45	5,49	4,50	5,50	5,49
24	4,07	4,69	4,72	4,67	4,66	4,66
28	4,50	4,16	4,27	4,27	4,13	4,10
32	5,08	4,34	4,15	4,58	3,85	3,89
36	6,14	4,72	5,03	4,95	5,16	5,27
40	6,65	5,22	5,25	5,28	5,25	5,22
44	7,02	5,25	5,23	5,42	5,19	5,24
48	7,19	5,45	5,31	5,23	5,17	5,39

Tabla 7.6: Tiempos de ejecución de la imagen 7.1b con un tamaño de 3000x3000

Como se puede observar en la tabla 7.6 el mejor resultado se obtienen con 32 *threads* y a partir de los 32 *locks*. El mejor resultado es de 3,85 segundos, con un *speed-up* de 5,7 y una eficiencia del 18,0 %.

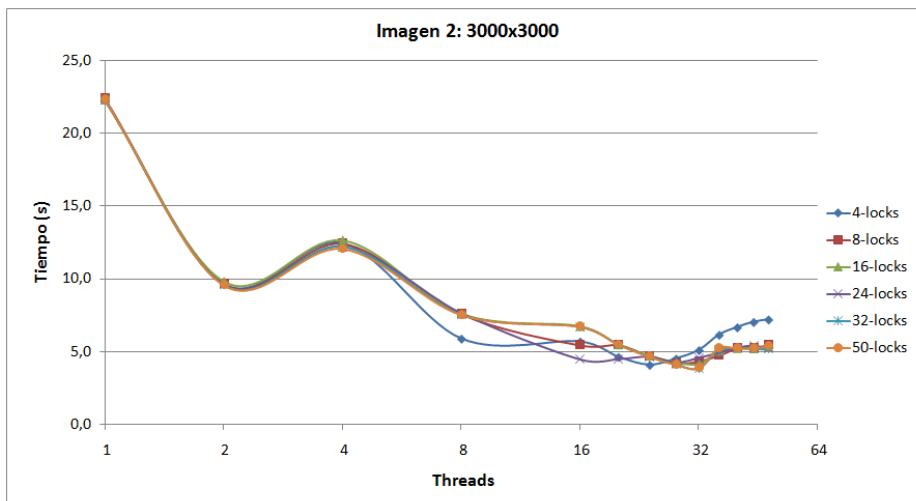


Figura 7.8: Gráfica de los tiempos de ejecución de la imagen 7.1b con un tamaño de 3000x3000

Resultado

Los resultados de la segmentación de esta segunda imagen han sido los que se muestran en 7.9, donde se ha marcado la lista L_{out} de color rojo y la lista L_{in} de azul. Se ha conseguido obtener el contorno las islas que es lo que se esperaba.

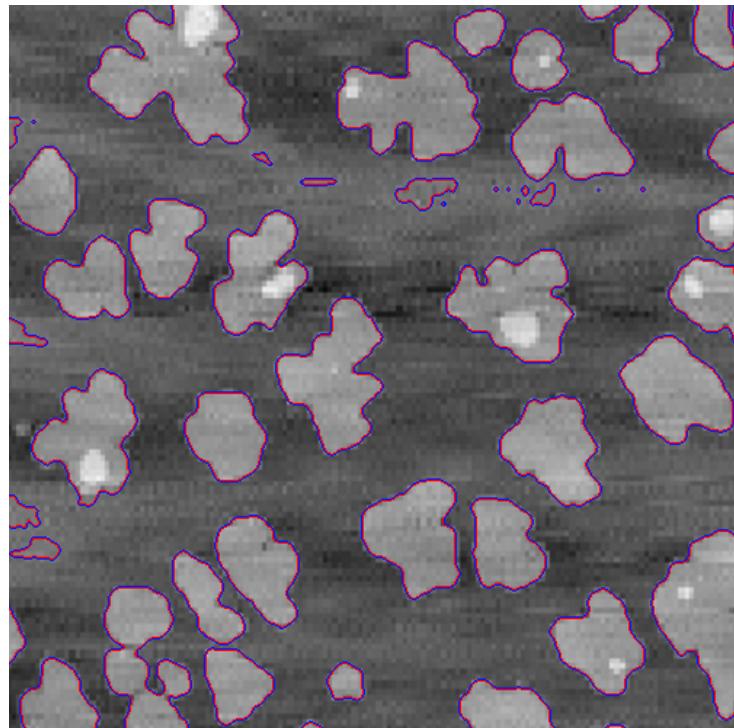


Figura 7.9: Resultado de la segmentación de la imagen 7.1b

7.3. Tercera imagen

La tercera imagen a probar será la imagen 7.1c y a continuación se presentan los resultados de las ejecuciones con diferentes tamaños de la imagen.

Tamaño 500x500

Threads \ Locks	4	8	16	24	32	50
1	0,72	0,73	0,72	0,73	0,72	0,72
2	0,48	0,48	0,48	0,48	0,48	0,48
4	0,34	0,34	0,34	0,35	0,33	0,33
8	0,21	0,28	0,28	0,27	0,27	0,27
16	0,20	0,19	0,24	0,16	0,24	0,24
20	0,17	0,20	0,20	0,16	0,20	0,20
24	0,15	0,17	0,17	0,17	0,17	0,17
28	0,17	0,15	0,15	0,15	0,15	0,15
32	0,17	0,14	0,14	0,14	0,14	0,14
36	0,18	0,14	0,13	0,13	0,13	0,13
40	0,19	0,12	0,13	0,12	0,12	0,12
44	0,19	0,14	0,12	0,12	0,12	0,12
48	0,22	0,18	0,14	0,14	0,13	0,17

Tabla 7.7: Tiempos de ejecución de la imagen 7.1c con un tamaño de 500x500

Como se puede observar en la tabla 7.7 los mejores resultados se consiguen con 40 y 44 *threads* y con mas de 8 *locks*, logrando así un tiempo de 0,12 segundos, un *speed-up* de 5,8 y una eficiencia del 14,6 %.

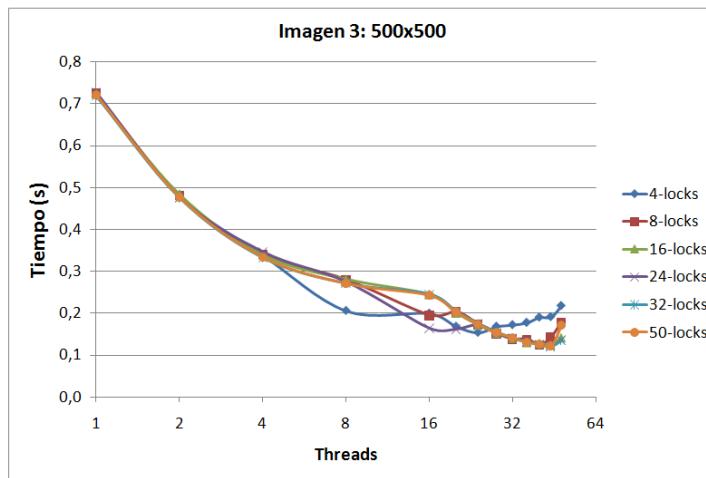


Figura 7.10: Gráfica de los tiempos de ejecución de la imagen 7.1c con un tamaño de 500x500

Tamaño 1500x1500

Threads \ Locks	4	8	16	24	32	50
1	7,67	7,71	7,67	7,68	7,69	7,66
2	4,78	4,79	4,76	4,77	4,75	4,75
4	3,43	3,45	3,44	3,40	3,38	3,37
8	1,97	2,81	2,80	2,71	2,72	2,70
16	1,86	1,82	2,31	1,46	2,30	2,30
20	1,56	1,88	1,87	1,47	1,86	1,86
24	1,33	1,57	1,57	1,56	1,58	1,57
28	1,42	1,37	1,39	1,39	1,39	1,38
32	1,45	1,26	1,26	1,26	1,28	1,30
36	1,58	1,19	1,20	1,26	1,24	1,18
40	1,69	1,35	1,33	1,33	1,33	1,33
44	1,73	1,33	1,31	1,31	1,30	1,31
48	1,79	1,34	1,33	1,34	1,33	1,31

Tabla 7.8: Tiempos de ejecución de la imagen 7.1c con un tamaño de 1500x1500

La mejor respuesta temporal en este caso se consigue mediante la combinación de 44 *threads* y 32 *locks*. Se obtiene así un tiempo de 1,30 segundos, un *speed-up* de 5,9 y una eficiencia del 13,5 %.

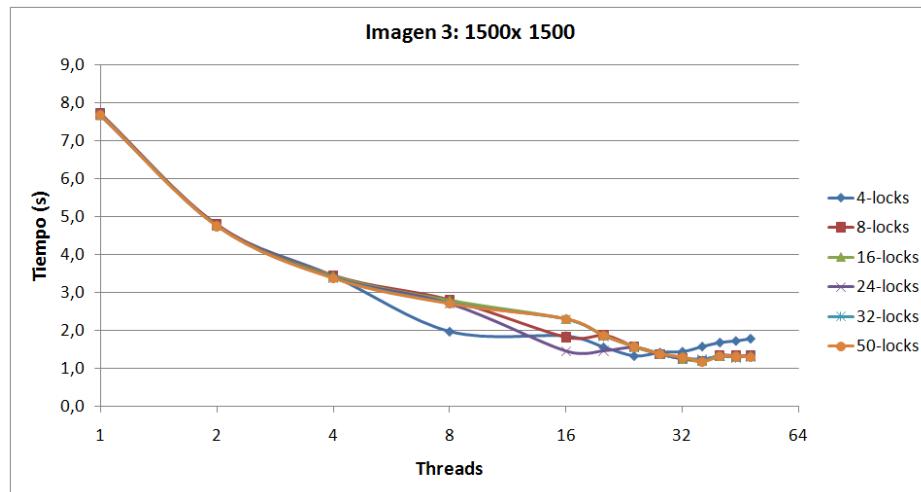


Figura 7.11: Gráfica de los tiempos de ejecución de la imagen 7.1c con un tamaño de 1500x1500

Tamaño 3000x3000

Threads \ Locks	4	8	16	24	32	50
1	30,89	30,78	30,84	30,88	30,77	30,91
2	13,23	13,91	13,19	13,20	13,43	13,09
4	16,18	16,28	16,33	16,10	16,00	15,88
8	6,78	9,48	9,54	9,46	9,44	9,48
16	6,82	6,68	8,53	5,32	8,46	8,50
20	5,55	6,77	6,76	5,42	6,84	6,85
24	4,81	5,70	5,69	5,67	5,74	5,74
28	4,89	4,92	4,96	4,90	4,97	4,96
32	4,71	4,44	4,51	4,45	4,48	4,46
36	5,21	4,88	4,91	5,01	4,66	4,61
40	5,58	5,02	5,14	5,00	5,02	5,08
44	5,69	5,03	4,90	4,92	4,91	4,98
48	5,96	4,98	4,84	4,90	5,01	4,93

Tabla 7.9: Tiempos de ejecución de la imagen 7.1c con un tamaño de 3000x3000

Como se puede observar en la tabla 7.9 el mejor resultado se consigue con 32 threads y 8 locks, logrando un tiempo de 4,44 segundos, un speed-up de 6,9 y una eficiencia del 21,7 %.

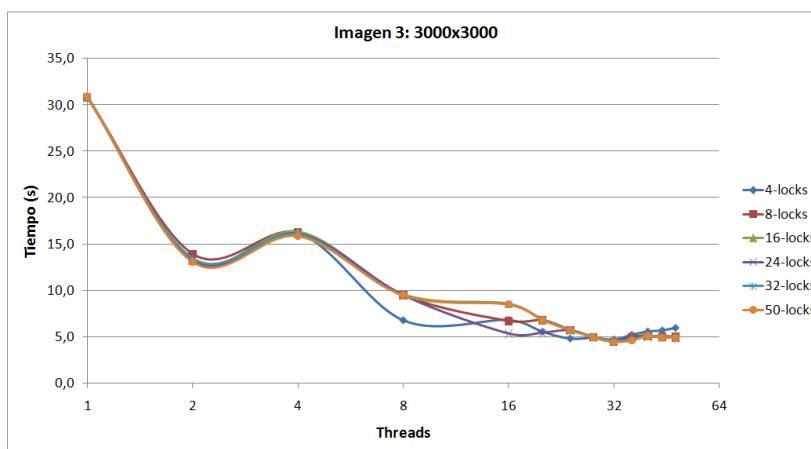


Figura 7.12: Gráfica de los tiempos de ejecución de la imagen 7.1c con un tamaño de 3000x3000

Resultado

Los resultados de la segmentación de esta tercera y última imagen han sido los que se muestran en 7.13, donde se ha marcado la lista L_{out} de color rojo y la lista L_{in} de azul. Se ha conseguido obtener el contorno las islas que es lo que se esperaba.



Figura 7.13: Resultado de la segmentación de la imagen 7.1c

Capítulo 8

Gestión del proyecto

8.1. Gestión del alcance

8.2. Gestión del tiempo

8.3. Gestión de los riesgos

El único riesgo más aparente que se puede producir en este proyecto es la pérdida de la memoria escrita o de la implementación paralela realizada a lo largo del proyecto. Para evitar que pueda producirse tal situación se requiere el uso de dos herramientas *online*: el sistema de almacenamiento de Google, *Drive*, y un sistema de gestión de versiones, en este caso *Github*. Con estas dos herramientas se ha podido realizar un sistema de *backup* sencillo como gestión de este riesgo. La memoria con todos sus documentos se ha guardado diariamente en el *Drive* de manera que en caso de pérdida se pueda recuperar los avances de un día anterior. La implementación se ha ido creando en manera de versiones en las que se le han ido añadiendo funcionalidades extras en cada versión. Cada una de estas versiones se ha ido subiendo a la plataforma *Github* del autor de esta memoria en [20].

8.4. Gestión del costes

Los costes de este proyecto han sido únicamente la dedicación de horas humanas con las que se ha realizado. Esto se puede ver en la tabla ??.

Capítulo 9

Conclusiones

CONCLUSIONES

9.1. Líneas futuras

A pesar de haber conseguido buenos resultados temporales de la paralelización del algoritmo *level set* estos se podrían haber mejorado un poco más de haber tenido más tiempo para la realización del proyecto. A continuación se listan unas posibles mejoras o ideas futuras interesantes:

1. Eliminar la compartición falsa en la memoria caché de algunas variables de la implementación que han tenido que ser creadas para poder realizar la paralelización. Por lo tanto, al ejecutar en una máquina SMP, como ciertos *cores* comparten entre ellos la misma memoria caché, al realizar escrituras sobre la misma variable invalida a los demás *cores* esa misma variable, por lo que tendrán que volver a cogerla de la memoria principal. Si esta operación se da muchas veces, puede suponer un sobrecoste alto a la implementación.
2. La idea propuesta de la realización de la paralelización de los *frames* de un vídeo puede llevarse a cabo mediante la librería MPI. Cada *frame* se le pasaría a cada nodo del *cluster* y que cada uno segmente esa imagen. En esta tarea se podrá utilizar el algoritmo desarrollado a lo largo de este proyecto en OpenMP. De esta manera, la combinación de estas dos técnicas conseguiría poder realizar la segmentación de un vídeo satisfactoriamente.
3. El trabajo realizado en este proyecto también se podía haber desarrollado con CUDA para realizar el algoritmo en GPUs. Sin embargo, esta

opción no se llevó a cabo ya que el tratamiento de las listas en CUDA es bastante complejo y no es «natural» comparado con el modelo de programación que se sigue en CUDA. Existen técnicas para realizar un tratamiento de estas listas más eficientemente aunque el hecho de que este tratamiento ya tenga complicación no augura muy buena eficiencia. Aparte de esto, el acceso a máquinas de pruebas podía suponer alguna complicación. Por todo ello se decidió no utilizar CUDA en ese primer momento aunque se cree que puede ser una buena opción el poder realizar este trabajo en un futuro.

Bibliografía

- [1] IEEE Xplore Digital Library. <http://ieeexplore.ieee.org/>.
- [2] L. Suta, F. Bessy, C. Veja, and M. F. Vaida. Active contours: Application to plant recognition. *Intelligent Computer Communication and Processing (ICCP), 2012 IEEE International Conference on*, pages p. 181 – 187, 2012.
- [3] F. Bessy. *Open, Fast and Efficient Level set Implementation*, 2012-2013. François Rabelais University's computer science laboratory, <https://code.google.com/p/ofeli/>.
- [4] E. Sánchez Izquierdo. *Concepción, diseño e implementación de un software multinivel para el servicio Morfokinetics*, 2015. Universidad del País Vasco.
- [5] Terry S. Yoo. *Insight into Images: Principles and Practice for Segmentation, Registration, and Image Analysis*. A K Peters, Ltd, 2004.
- [6] Yu-Jin Zhang. *Advances in Image and Video Segmentation*. 2006.
- [7] Kevin McGuinness. *Image Segmentation, Evaluation, and Applications*. PhD thesis, Dublin City University, 2010. p. 11-17.
- [8] B. Basavaprasad and M. Ravi. A comparative study on classification of image segmentation methods with a focus on graph based techniques. *International Journal of Research in Engineering and Technology*, vol. 3, 2014.
- [9] Arnau Oliver Malagelada. *Automatic Mass Segmentation In Mammographic Images*. PhD thesis, University of Girona, <http://eia.udg.edu/~aoliver/publications/tesi/>, 2008. chap. 2.3.3.
- [10] E. Smistad, T. Falch, M. Bozorgi, A. Elster, and F. Lindseth. Medical image segmentation on GPUs – A comprehensive review. *Med. Image Anal.*, vol. 20, 2015. p. 1-18.

- [11] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, pages p. 321–331, 1988.
- [12] Y. Shi and W. Karl. A fast implementation of the level set method without solving partial differential equations. 2005. <http://iss.bu.edu/yshi>.
- [13] A. Hagan and Y. Zhao. *Parallel 3D Image Segmentation of Large Data Sets on a GPU Cluster*, volume vol. 5876. Springer Berlin Heidelberg, 2009.
- [14] H. Mostofi and K. College. *Fast Level Set Segmentation of Biomedical Images using Graphics Processing Units*, 2009. Final Year Project, University of Oxford, <https://code.google.com/p/cudaseg/>.
- [15] M. Jeon, M. Alexander, and N. Pizzi. Parallel image segmentation with level set methods. *Journal of Parallel and Distributed Computing*, 2015.
- [16] M. R. Hajihashemi and M. EI-Shenawee. MPI parallelization of the level-set reconstruction algorithm. *Proc. IEEE Int. Symp. Antennas Propag./URSI Nat. Radio Sci. Meeting*, pages p. 1 – 4, 2009.
- [17] T. Chan and L. Vese. Active contours without edges. vol. 10:p. 266–277, 2001. <https://code.google.com/p/ofeli/>.
- [18] OpenMP API specification for parallel programming. <http://openmp.org/>.
- [19] Livermore Computing Center (LC). <https://computing.llnl.gov>.
- [20] D. Franco. Implementación paralela. 2015. <https://github.com/dfranco007/parallelOfeli>.