Trabajo de Fin de Grado

Grado en Ingeniería Informática
Computación

Segmentación de imágenes usando transformadas de distancia

Autor: Eloi Sevilla López
Director: Joan Climent Vilaró
Tutor de GEP: Marfil Sanchez Fernando

Índice

1. Introducción y contextualización

La segmentación de imágenes es un proceso del campo de la visión artificial, que consiste en separar los elementos de una imágen en varios grupos. Cada grupo puede simbolizar cosas distintas dependiendo del contexto en el que se encuentre. Por ejemplo, en una aplicación para una fábrica en la que se empaqueta arroz, y tenga cómo función controlar la calidad de los granos de arroz, se podría utilizar un sistema de segmentación de imágenes tal que, dada una fotografía de varios granos de arroz, se segmente la imágen para aportar a un sistema informático la información necesaria para contar los granos de arroz, calcular el grosor medio de los granos, la forma que tienen, etc. Otro ejemplo un poco más complejo de segmentación de imágenes podría ser un sistema de seguridad, encargado de de detectar caras en un vídeo captado por una cámara de seguridad y aislarlas de la resta de la imágen, para poder analizarlas y extraer sus características.

Dada la multitud de aplicaciones posibles para las diferentes técnicas de segmentación de imágenes, resulta interesante estudiar diferentes técnicas y algoritmos que permitan segmentar imágenes de forma rápida y eficiente, y que tengan en cuenta diferentes factores cómo la forma de los objetos, los colores, las texturas, etc. Por eso en este TFG se estudia un algoritmo para preprocesar imágenes en escala de grises con el objetivo de recibir datos útiles para poder segmentar dicha imágen. Más concretamente, el centro del estudio es un algoritmo introducido por S. Beucher, el cual devuelve la transformada de quasi-distancia de una imagen en escala de grises.

El algoritmo se ha implementado en forma de varias funciones de Matlab, para que sean fáciles de utilizar y se puedan comprobar los resultados y hacer experimentos con la simplicidad que aporta este software.

1.1. Objetivos del proyecto

Los objetivos principales del proyecto es programar el algoritmo de la quasi-distancia, y comparar su eficiencia y efectividad como herramienta de segmentación de imágenes en escala de grises con otras alternativas mediante experimentos. Se utilizan dos implementaciones del cálculo de la quasi-distancia, y se comparan estas dos implementaciones entre ellas. Una de estas dos implementaciones es la versión diseñada por S. Beucher, a la que llamaremos “Erosion-Based”, ya que su funcionamiento se basa en aplicar erosiones sobre la imagen. Por otra parte, la otra versión está diseñada por Raffi Enficiaud, y la llamaremos “Queue-Based” ya que su funcionamiento se basa en el uso de colas para calcular la quasi-distancia eficientemente.

1.1.1. Implementación del software

En este proyecto se han implementado un total de 4 algoritmos:

- Cálculo de Q Erosion-Based: El objetivo de este algoritmo es hacer calcular Q a partir de una imagen en escala de grises. Este resultado es un resultado intermedio para la quasi-distancia, que se tiene que normalizar ya que no sigue la propiedad de 1-Lipschitz.
- Cálculo de RQ Erosion-Based: Este algoritmo calcula RQ a partir de Q. Básicamente este es el algoritmo regularizador, la regularización consiste en forzar la imagen Q para que siga la propiedad de 1-Lipschitz. Esta imágen es la quasi-derivada.
- Cálculo de Q Queue-Based: Este algoritmo también calcula Q a partir de una imagen en escala de grises, pero lo hace usando otras estructuras de datos y utilizando distintas funciones. Este algoritmo se implementa más que nada para comparar esta versión con la Erosion-Based.
- Cálculo de RQ Queue-Based: Al igual que con el algoritmo anterior, este también se implementa con el objetivo principal de comparar el tiempo de ejecución de éste y de la versión Erosion-Based. Calcula RQ a partir de Q.

1.1.1.1. Formato

Estos algoritmos se han implementado en forma de función de Matlab, para que sean sencillos de ejecutar y para poder comparar de manera práctica las dos versiones de los algoritmos. Al estar implementado de esta manera, se pueden utilizar de manera aislada, y se podría por ejemplo calcular Q con el algoritmo Erosion-Based y regularizar el resultado utilizando el algoritmo Queue-Based.

1.1.1.2. Requisitos del sistema

Los requisitos que siguen las funciones son principalmente los siguientes:

- El output de las funciones es siempre una matriz representable en forma de imagen en escala de grises.
- Las funciones son intuitivas y fáciles de utilizar en un programa, y los resultados son consistentes en su formato. La resolución de la imágen resultado es siempre la misma resolución que la de la imágen de entrada.
- Las funciones aportan el resultado esperado siempre y cuando la entrada siga los pre-requisitos, independientemente de factores cómo la resolución de la imágen.
- Las funciones son lo más eficientes posible, y siempre siguiendo la especificación del paper de referencia.

1.1.2. Realización de experimentos

La otra parte de este TFG consiste en realizar diferentes experimentos para comparar la eficiencia y eficacia de los algoritmos implementados. Se han hecho varios experimentos con esa finalidad, y se han comparado los resultados de las funciones con resultados que aportan alternativas similares.

1.2. Definición de conceptos propios del tema

A continuación se definen varios conceptos clave que se mencionan a lo largo de este proyecto y qué son completamente necesarios para la comprensión de éste:

1.2.1. Imagen Binaria

Imagen representable como una matriz con solamente 2 posibles valores. Una manera común de representar una imagen binaria es una matriz de booleanos en la que un elemento con valor “true” representa un pixel blanco, y un elemento con valor “false” representa un pixel negro.

1.2.2. Transformada de distancia

La transformada de distancia [1] es un operador que se suele usar para operar con imágenes binarias. El resultado de aplicar este operador es una imagen en escala de grises, cuyos valores indica cómo de lejos está un pixel del borde del objeto. Por ejemplo, en la siguiente figura [Fig 1], se ve como los píxeles más interiores de las figuras se quedan con un valor más alto de blanco. Este operador resulta muy útil para actividades como el recuento de objetos, ya que permite separar dos objetos unidos por pequeñas estructuras o parcialmente superpuestos, ya que en esos casos, el resultado tendrá un máximo en cada objeto distinto. Este operador funciona muy bien para objetos simples como los de la figura, pero suele dar resultados problemáticos con imágenes más complejas.

Original				
Transformada de distancia				

Fig.1. Ejemplo de la transformada de distancia euclídea aplicada a varias imágenes binarias. En la última se puede ver como la transformada tiene dos máximos, de los que se puede interpretar que son dos objetos unidos y la franja negra es la separación entre ellos. Fuente:<https://homepages.inf.ed.ac.uk/rbf/HIPR2/distance.htm>
[Consulta: 20 febrero 2020]

1.2.3. Transformada de quasi-distancia

La transformada de la quasi-distancia [2] es un algoritmo, introducido por S. Beucher, que funciona de manera análoga a la transformada de distancia pero en imágenes en escala de grises, y será el principal objeto de nuestro estudio.

1.2.4. Watershed

El watershed [3] es una técnica de segmentación de imágenes que interpreta la imagen como si fuese un relieve topográfico e inicia una inundación desde varios puntos. En las fronteras donde se unen varias inundaciones, se crea una pared que representa el borde entre varios segmentos de la imagen.

1.2.5. Continuidad de Lipschitz

La continuidad de Lipschitz [4] es una propiedad que limita la pendiente que puede tener una función. En este proyecto mencionaremos varias veces la propiedad 1-Lipschitz, que limita la pendiente a 1. Ya que aplicamos la propiedad a imágenes, calcularemos la pendiente comparando píxeles vecinos, es decir, para qué una imágen siga la propiedad 1-Lipschitz se tiene que cumplir la siguiente propiedad, siendo p el valor de un píxel cualquiera y v el valor de un píxel vecino de p:

$$\forall p \neg \exists v \mid v - p > 1$$

1.2.6. Erosión

En procesado de imágenes, la erosión es un operador que puede utilizarse tanto en imágenes binarias cómo en escala de grises. En el caso binario, se puede visualizar la erosión como el resultado de pasar un elemento sobre la imágen. Este elemento únicamente puede pasar por las zonas donde la imágen sea de color blanco. El resultado de la erosión será blanco en los píxeles por donde ha podido pasar el elemento, y 0 en las zonas donde no ha podido pasar. A continuación se muestra una imágen de ejemplo [Fig 2].

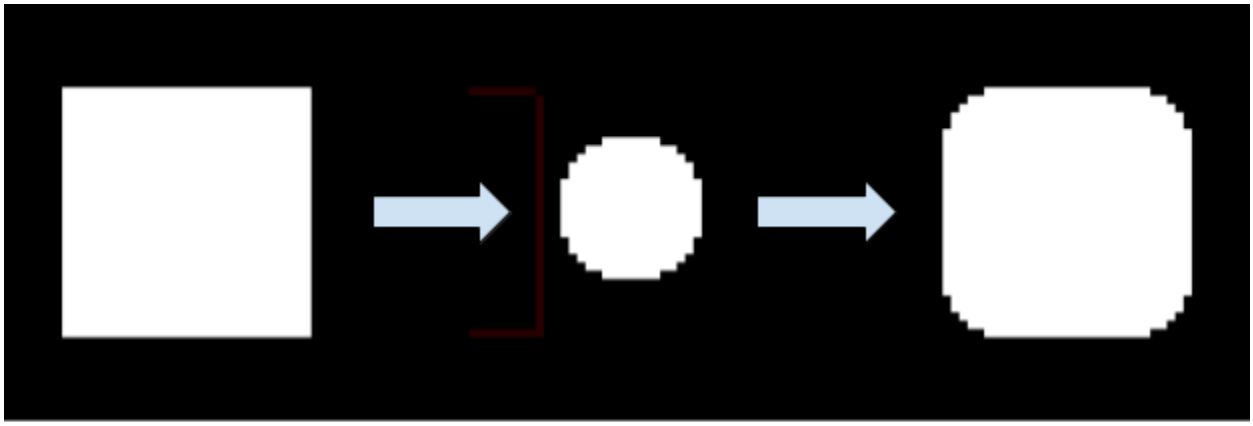


Fig 2: de izquierda a derecha: imagen original, elemento utilizado para la erosión, imagen resultante de la erosión. Elaboración propia.

En el caso de la erosión en imágenes en escala de grises, el proceso es el mismo, pero se añade una dimensión adicional: las dos coordenadas de la imágen y el valor de gris cómo tercera dimensión. El elemento en este caso también tiene tres dimensiones, y únicamente puede pasar por debajo de la superficie que crea la imagen.

1.2.7. Hierarchical Queue

Una hierarchical queue [5] es una estructura de datos que añade funciones adicionales a las colas tradicionales. De manera intuitiva una hierarchical queue se puede visualizar cómo varias colas, cada una correspondiente a una prioridad concreta. Al insertar un elemento en la hierarchical queue, se puede decidir a qué prioridad insertar dicho elemento, lo que permite extraer los elementos de la cola en un orden más específico que extraer los elementos en el mismo orden en que son insertados.
Por ejemplo, al insertar un elemento que se sabe que se va a necesitar extraer pronto en una hierarchical queue, se puede insertar en la cola con máxima prioridad. De este modo, el elemento recién insertado solamente tendrá por delante los elementos insertados anteriormente con la misma prioridad. En una cola tradicional no podría hacerse esto, el elemento tendría por delante todos los elementos insertados antes que él, ya que no se pueden asignar prioridades.

1.3. Problema a resolver

El problema a resolver es la dificultad de segmentar imágenes en escala de grises utilizando algoritmos tradicionales. Por eso, en este proyecto se implementan funciones para calcular la quasidistancia, con el objetivo de conseguir una segmentación decente en un tiempo aceptable. Por ese motivo también se hacen diferentes experimentos, con el objetivo de demostrar que estos algoritmos son capaces de conseguir una segmentación mejor que opciones similares, en un tiempo decente. También se comparan las versiones Erosion-Based y Queue-Based, para ver qué versión se ajusta mejor a nuestras necesidades

1.3.1 Opciones similares

A continuación se tratan opciones similares con las que se comparan los algoritmos implementados:

1.3.1.1. Watershed sobre la imagen en escala de grises

A veces resulta útil aplicar el algoritmo de watershed directamente sobre la imagen en escala de grises [Fig 3], sin ningún preprocesado. Si ese es el caso se empieza la inundación en los puntos en los que la imágen tiene valores mínimos. Utilizando este método, la imágen será segmentada por las zonas donde haya picos en los valores de gris.
Si se visualiza la imágen como si fuera un mapa de relieves, con valores de gris más altos indicando relieves más altos, el resultado de esta técnica separaría los valles, creando fronteras entre las montañas que los separan.

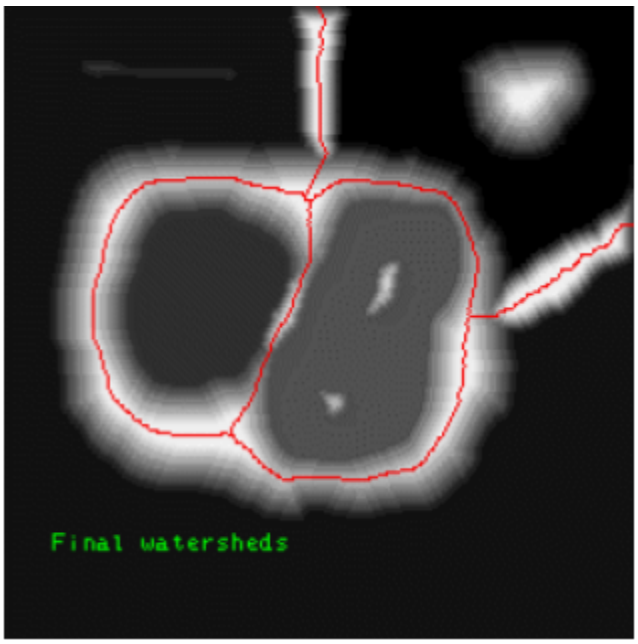


Fig.3. Ejemplo de watershed sobre imagen en escala de grises. Fuente:<http://www.cmm.mines-paristech.fr/~beucher/wtshed.html> [Consulta: 22 febrero 2020]

1.3.1.2. Watershed sobre la imagen gradiente

Este método consiste en calcular la imágen gradiente antes de aplicar el watershed. De esta manera, la imagen debería ser segmentada por las zonas con valores de gris similares. Usando este método, es muy común que los resultados tengan mucha sobre-segmentación [Fig 4]. Si interpretamos la imagen original como un mapa de relieves, las fronteras entre los segmentos esta vez se crearían en las zonas donde cambia la altura del relieve de manera brusca.

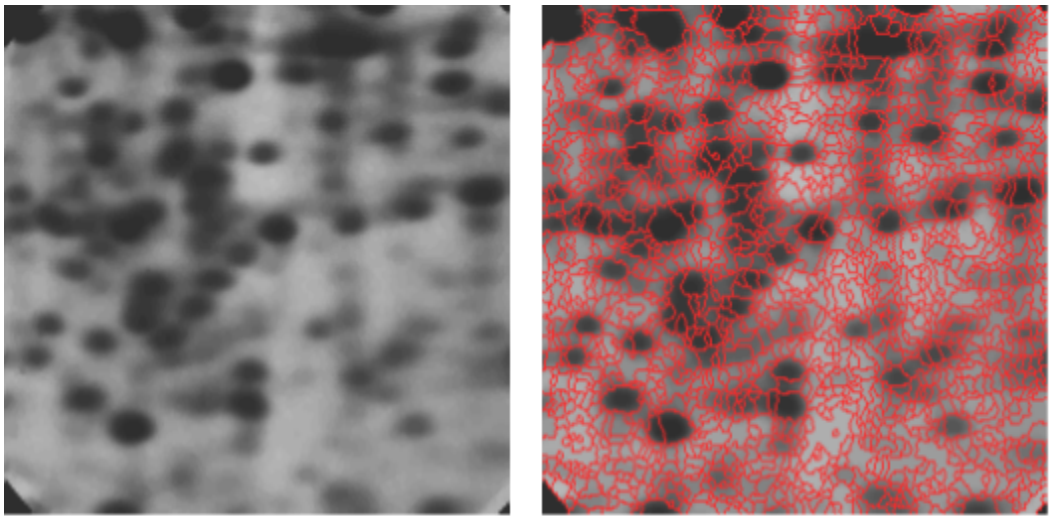


Fig.4. Ejemplo de watershed sobre imagen gradiente. Cómo se puede apreciar en este ejemplo, esta técnica tiende a dar problemas de sobre-segmentación. Fuente:<http://www.cmm.mines-paristech.fr/~beucher/wtshed.html>
[Consulta: 22 febrero 2020]

En la siguiente figura[Fig 5] , se pueden apreciar los pasos de este proceso, y cómo al fin y al cabo los resultados del watershed son distintos gracias a que se aplica sobre una imagen distinta a la original.

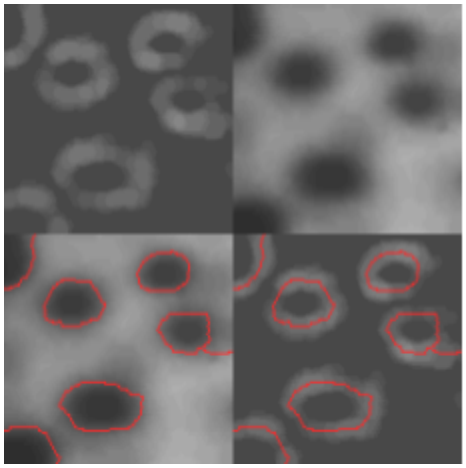


Fig.5. Ejemplo de los pasos de aplicar watershed sobre la imagen del gradiente. De arriba a abajo y de izquierda a derecha: Imagen original, gradiente de la imagen, resultado del watershed sobre el gradiente, resultado del watershed superpuesto a la imagen original.
Fuente:<http://www.cmm.mines-paristech.fr/~beucher/wtshed.html>
[Consulta: 22 febrero 2020]

1.3.1.3. Watershed con markers

Este método consiste en iniciar la inundación en marcas creadas previamente al watershed [Fig 6]. La fiabilidad de esta técnica depende de lo bien que estén creadas las marcas. Es decir, ya que, a diferencia de la técnica anterior, en esta técnica aplicamos el watershed en la imágen original, y lo que realmente se hace de manera distinta es el watershed en sí mismo, ya que en lugar de empezar por los valles en la imágen, esta vez tiene unos puntos predeterminados donde empieza, y la resta del algoritmo es completamente igual.
Una manera automática de crearlas es utilizando el algoritmo que se implementa en el TFG y creando marcas en los picos de la transformada de quasi-distancia, pero hay infinidad de métodos, que son más o menos apropiados dependiendo del contexto de la imagen. En ocasiones los métodos que mejores resultados dan en una imágen, son los que peor funcionan en otra imágen, por lo tanto la alta variedad en la calidad de los resultados podrían considerarse un punto débil del watershed con markers.

Fig.6. Ejemplo de watershed usando markers. De izquierda a derecha: Imagen original, imagen con marcas hechas (a mano), imagen segmentada. En este ejemplo el algoritmo ha dado unos resultados muy buenos, debido principalmente a que las marcas se hicieron a mano. Fuente:<http://www.cmm.mines-paristech.fr/~beucher/wtshed.html>
[Consulta: 22 febrero 2020]

1.3.1.4. Binarización de la imagen y watershed sobre la distancia

Otro método disponible consiste en binarizar la imagen, calcular la transformada de distancia tradicional sobre la binarización, y finalmente implementar watershed sobre la transformada de distancia. El principal problema de este método es el primer paso: la binarización. Básicamente, resulta complicado binarizar la imagen de manera consistente, existen varios métodos para binarizar una imagen, pero los resultados de estos métodos suelen variar bastante dependiendo de las características de la imagen

1.4. Gestión del proyecto

A continuación se habla sobre diferentes elementos sobre la gestión del proyecto.

1.4.1. Método de trabajo

La metodología de trabajo que se ha seguido durante el proyecto es el método de desarrollo iterativo [6]. Este método consiste en planificar el proyecto en varios bloques temporales, llamados iteraciones. A continuación, en cada iteración se hace una versión funcional del proyecto, de manera que este va evolucionando a lo largo del tiempo, cada vez con nuevas funcionalidades o con elementos de versiones anteriores mejorados. De esta manera, el proyecto va evolucionando a lo largo del tiempo, y se minimiza el riesgo de que quede algún elemento importante del proyecto para el final, ya que siempre se añadirán funciones al proyecto en función de la prioridad que tengan.

1.4.2. Definición de las tareas

Las tareas que definen el proyecto han cambiado un poco desde el inicio del proyecto. Al final se ha hecho un mayor énfasis sobre la experimentación de los algoritmos mediante distintos tests. El cambio que esto representa es un incremento sobre las horas de analista, y un decremento sobre las horas de programador. La siguiente lista resume las tareas del proyecto:

Tarea	Tiempo (h)	Horas por rol (h)		
		Jefe proyecto	Programador	Analista
Gestión del proyecto	65	65		
Definición del contexto y alcance	25	25		
Planificación temporal	10	10		
Presupuesto y sostenibilidad	10	10		
Integración del documento final	20	20		
Trabajo previo	30	30		
Búsqueda de información	20	20		
Estudio de alternativas	10	10		
Primera versión	82		82	
Programación de las funciones básicas	72		72	
Lectura de imágenes	2		2	
Cálculo de la transformada de la quasi-distancia	40		40	
Segmentación de la imagen	25		25	
Impresión del resultado	5		5	
Corrección de errores	10		10	
Segunda versión	45		15	30
Optimización del código	15		15	
Análisis de eficiencia	30			30
Experimentos	80		20	60
Diseño de experimentos	30			30
Programación de experimentos	20		20	
Análisis y redactado de los resultados	30			30
Documentación	115	115		
Redacción de la memoria	75	75		
Preparación de la presentación oral	40	40		
Total	417	210	117	90

Fig 7: Horas de proyecto. Fuente: Elaboración propia

1.4.3. Presupuesto

El presupuesto teórico del proyecto se ha calculado de manera previa al proyecto. Se estimó en su momento un presupuesto total de 13473.2 €. Debido al cambio en las tareas, los costes humanos del proyecto varían un poco respecto a la planificación inicial.
Para calcular el nuevo presupuesto, utilizamos los siguientes sueldos, estimados a través de la página web "www.payscale.com" [7].

Rol	Sueldo
Jefe de proyecto	30 €/h
Programador	25 €/h
Analista	20 €/h

Fig 8: Sueldos. Fuente: Elaboración propia

Entonces, podemos calcular el nuevo coste por personal:

Rol	Horas (h)	Coste (€)
Jefe proyecto	210	6300
Programador	117	2925
Analista	90	1800
Total	417	11025

Fig 9: Costes humanos. Fuente: Elaboración propia

A estos costes humanos, tenemos que añadir los costes indirectos y los posibles costes causados por imprevistos. Los costes indirectos suman 380€ y son principalmente la amortización del ordenador, el espacio de trabajo, consumo eléctrico y el consumo de internet. Los costes por imprevistos estan resumidos en la siguiente tabla [Fig. 10].

Imprevisto	Precio	Riesgo	Coste
Contingencia	11025 + 380	15%	1710.75€
Nuevo Pc	1200 €	5%	60 €
Incremento del tiempo de programación (50 h)	1250 €	10%	125 €
Total			1895.75 €

Fig 10: Gastos imprevistos. Fuente: Elaboración propia

Así pues, el cálculo del nuevo coste total se resume en la siguiente tabla [Fig. 11]:

Actividad	Coste
Costes humanos	11025 €
Costes indirectos	380 €
Contingencia + Imprevistos	1895.75 €
Total	13300.75 €

Fig 11: Coste total. Fuente: Elaboración propia

1.4.4. Sostenibilidad

A continuación se discute la sostenibilidad del proyecto, desde un aspecto económico, ambiental y social.

Desde un punto de vista económico, este proyecto tiene la mayoría de costes en el aspecto humano, ya que al ser un software que únicamente requiere un ordenador para ser ejecutado, los costes indirectos de éste son prácticamente nulos comparados con los costes producidos a causa de su programación. Esto hace que reducir costes sea difícil, ya que la única manera de reducirlos de manera importante sería decrementar las horas de trabajo en éste, y eso podría llevar a un resultado de menor calidad.

Teniendo lo anteriormente mencionado en cuenta, opino que económicamente este proyecto tiene una sostenibilidad bastante alta, ya que la mayoría de su coste viene de la puesta en producción y de tal manera que no es trivial reducirlo. El coste durante su vida útil es prácticamente nulo, ya que en principio sólo entrarían en juego los costes indirectos.

Respecto al impacto ambiental, tanto durante la puesta en producción como durante su vida útil son mínimos, ya que se trata de un software que solo requiere un ordenador con matlab instalado para funcionar.

Cómo consecuencia, la vída útil del proyecto es básicamente infinita, ya que nunca se deteriorará ni dejará de funcionar. La única posibilidad que podría reducir la vida útil de este producto sería una mejora en el hardware o el software implicado que diera pie a posibles mejoras en su eficiencia, y por tanto, la posibilidad de una nueva versión con mejor eficiencia, pero eso no haría que la versión antigua dejase de funcionar.

Finalmente, desde un punto de vista social, mientras que no existe una necesidad imperativa de la realización del proyecto en la actualidad, opino que es buena idea realizarlo, ya que puede llevar a nuevos sistemas de visión por computador que no son accesibles con la tecnología existente actualmente.

A parte de lo mencionado anteriormente, a nivel personal la realización del proyecto me ha aportado un mayor nivel de conocimientos sobre las técnicas de segmentación de imágen, y estos conocimientos podrán ser transmitidos a cualquier persona interesada en el proyecto ya que los resultados están redactados en la memoria.

2. Información técnica

A continuación se da información técnica sobre el funcionamiento de los algoritmos implementados en este proyecto para calcular la quasi-distancia. Se empezará hablando del resultado esperado de los algoritmos, para poder usar ese conocimiento cómo base para comprender el funcionamiento de los algoritmos y posteriormente su implementación.

2.1. Resultado esperado

El resultado deseado después de ejecutar estos algoritmos es la quasi-distancia, el equivalente de la transformada de la distancia para imágenes en escala de grises, así que debe cumplir ciertas características. Primero, los píxeles completamente negros tienen que seguir siendo negros en la imagen resultante, y cuanto más pequeños sean los objetos, menor valor tendrán los píxeles del resultado de la quasi-distancia correspondientes a ese objeto, ya que el valor de estos píxeles representa la distancia que tienen los píxeles en la imagen original hasta el píxel negro más cercano. Hay que mencionar que dependiendo de la implementación, la transformada de la distancia también podría medir la distancia hasta un pixel blanco. Ambas implementaciones son correctas, y para pasar de una a otra solamente es necesario invertir la imagen.

Otra característica que debe cumplirse es que todos los píxeles han de cumplir la propiedad de 1-Lipschitz. Esto se puede visualizar cómo si los valores de gris representaran alturas distintas en un terreno. En ese caso, el seguir la propiedad de 1-Lipschitz significa que en ningún momento habrá barrancos o zonas con mucha pendiente: todas las zonas del terreno tendrán cómo máximo una pendiente de 1 unidad.

Por último, otra característica que debe cumplir el resultado es que los valores de sus píxeles únicamente debe incrementar cuando haya algún cambio en los valores de grises, si no hay ninguna variación en la imagen en escala de grises, los valores del resultado tienen que ser 0. Estos resultados son muy útiles al aplicar watershed, ya que si lo miramos desde un punto de vista topográfico, estos son montes con su cúspide en el centro de un objeto, que van allanándose a medida que llegan al extremo del objeto. Si se invierte ese resultado, ahora se tiene el punto más bajo, y por tanto por donde empezará a aplicarse el watershed, en el centro del objeto y se formarán las fronteras del algoritmo en las fronteras del objeto.

2.2. Funcionamiento de los algoritmos

Para llegar hasta un resultado que siga las características mencionadas anteriormente, el método Erosion-Based utiliza dos algoritmos distintos. El primer paso consiste en ir haciendo la erosión a la imagen original en escala de grises hasta que se llegue a un punto en el que erosionar la imagen no varía la imágen. Después de cada erosión se compara la imagen antes y después de la erosión, y en los píxeles en los que hay una variación se incrementa el valor que tendrá el resultado de este algoritmo en el píxel con las mismas coordenadas. Esto produce una imagen cuyos píxeles tienen un valor más elevado en las zonas donde hay una variación en los valores de gris de las imágenes originales.

La imagen resultante del algoritmo anterior no es el resultado final, ya que no sigue la propiedad de 1-Lipschitz, así que se requiere un segundo algoritmo para hacer que sí que la siga. Este segundo algoritmo regulariza la imagen anterior decrementando el valor de todos los píxeles que no sigan esta propiedad. El método utilizado para esto es utilizar la erosión de la imagen, pero en este caso no se va erosionando paso a paso, sinó que se utiliza la erosión para encontrar los píxeles que no siguen la propiedad, se rectifican dichos píxeles para que sí que sigan la propiedad, y se va repitiendo esto hasta que todos los píxeles cumplen la propiedad. Una vez terminada esta regularización ya se ha logrado el resultado de la quasi-distancia.

Por otra parte, están los algoritmos Queue-Based. Estos algoritmos siguen la misma estructura general que los Erosion-Based, con la diferencia que intentan mejorar los tiempos de ejecución guardando la información sobre los píxeles que deberían procesarse en la siguiente iteración en lugar de erosionar la imagen entera. El primer algoritmo logra esto guardando los píxeles que tienen píxeles con mayor valor que ellos, y incrementando el valor de los píxeles correspondientes en el resultado. Este proceso simula el hecho de ir haciendo la erosión de la imágen, lo que hace que el resultado sea el mismo que el del primer algoritmo Erosion-Based.

Finalmente, el algoritmo regularizador Queue-Based utiliza una hierarchical queue para almacenar los píxeles que se tendrán que actualizar en la jerarquía correspondiente a su valor. De esta manera, se puede actualizar el valor de todos los píxeles en el orden más adecuado para no tener que actualizar el mismo valor más veces de las necesarias. La actualización de los píxeles se hace utilizando el mínimo valor de los píxeles vecinos, al igual que con la versión Erosion-Based. Al terminar la ejecución de este algoritmo se obtiene también la quasi-distancia.

3. Implementación

A continuación hablaremos sobre la implementación de los distintos algoritmos utilizados en este proyecto.

3.1. Pseudocódigo de los algoritmos

Este es el pseudocódigo de las funciones implementadas:

3.1.1. Cómputo de Q Erosion-Based

Entrada: I
Salida: Q, R

1. W1 ← I
2. Q, R, W2 ← 0
3. i ← 0
4. repeat
5. i ← i + 1
6. W2 ← ∈(W2)
7. residue ← W1 - W2
8. E ← (residue ≥ R and residue ≠ 0)
9. Q[E] ← ∨{i,Q[E]}
10. R ← ∨{residue, R}
11. W1 ← W2
12. Until residue = 0

Este algoritmo es el descrito por S.Beucher en el paper “Transformations résiduelles en morphologie numérique” [8], es el primer paso para calcular la quasidistancia. La entrada I es la imagen a procesar y las salidas Q y R son la quasidistancia sin regularizar y el residuo respectivamente.

3.1.2. Regularización de Q Erosion-Based

Entrada: Q
Salida. RQ

1. RQ ← Q
2. i ← 0
3. repeat
4. W ← RQ-∈(RQ)
5. E ← ⌐(W ≤ 1)
6. RQ[E] ← ∈(RQ[E]) + 1
7. until E = Ø

Este algoritmo, también disponible en “Transformations résiduelles en morphologie numérique” [8], tiene como función regularizar el resultado del algoritmo anterior para que sigua la propiedad de 1-Lipschitz. La entrada Q es la quasidistancia sin regularizar y la salida RQ es la quasidistancia regularizada.

3.1.3. Cómputo de Q Queue-Based

Entrada: I
Salida: Q, R

1. C ← candidate, W1 ← I, W2 ← I, Q ← 0, R ← 0
2. f1, f2 ← Ø
3. forall the p ∈ W1 do
4. forall the v ∈ {N(p, W1)\p} do
5. if (v < p) then f1 ← f1 + p, break
6. i ← 1
7. while f1 ≠ Ø do
8. forall the p ∈ f1 do
9. C(p) ← candidate
10. W2(p) ← ∧Np(W1)
11. residue ← W1(p) - W2(p)
12. if residue ≥ R(p) then R(p) ← residue; Q(p) ← i
13. forall the p ∈ f1 do
14. W1(p) ← W2(p)
15. forall the v ∈ Np(W2) do
16. if (v > p) and (C(v) ≠ in-queue) then f2 ← f2 + v; C(v) ← in-queue
17. f1 ← f2; f2 ← Ø; i ← i + 1

Este algoritmo descrito por Raffi Enficiaud en “Queue and Priority Queue Queue Based Algorithms for Computing the Quasi-distance Transform” [2] pretende mejorar la eficiencia del algoritmo Erosion-Based utilizando colas para procesar menos píxeles en cada iteración. Las entradas y salidas son igual que en la versión Erosion-Based, y f1 y f2 son colas para almacenar información sobre los píxeles a procesar.

3.1.4. Regularización de Q Queue-Based

Entrada: Q
Salida: RQ

1. RQ ← Q
2. C ← none
3. hq ← Ø
4. forall the p ∈ RQ do
5. forall the v ∈ Np do
6. if v > p + 1 then C(v) ← in-queue
7. forall the p ∈ RQ do
8. if C(p) = in-queue then
9. forall the v ∈ Np do
10. least-neighbor ← Q(v) + 1
11. if (C(v) = none) and (p > least-neighbor) then
12. RQ(p) ← least-neighbor
13. hq ← hq + p at priority least-neighbor
14. while hq ≠ Ø do
15. pr ← highest priority of hq
16. forall the p ∈ hq(pr) do
17. forall the v ∈ Np do
18. if RQ(v) > pr + 1 then hq ← hq + v at priority pr + 1, RQ(v) ← pr + 1
19. hq ← empty priority pr

Este algoritmo, también presentado por Raffi Enficiaud en “Queue and Priority Queue Queue Based Algorithms for Computing the Quasi-distance Transform” [2], pretende mejorar la eficiencia de la versión Erosion-Based utilizando la hierarchical queue hq.

3.2. Código en Matlab

A continuación se explorará el código de las diferentes funciones que fueron implementadas para calcular la quasidistancia. Nótese que en el apartado de resultados de cada función se ejecutan utilizando la imagen estándar de Lenna, únicamente para visualizar el resultado obtenido con las funciones, en esta ocasión no se está buscando realmente ninguna segmentación.

3.2.1. QErosionBased

3.2.1.1. Definición

Esta función computa la transformación de la quasidistancia de una imagen en escala de grises utilizando el algoritmo Erosion-Based.
Entrada:

- I: matriz en formato uint8 qué representa la imagen original en escala de grises.

Salida:

- Q: matriz en formato uint8 qué representa la transformada de la quasidistancia antes de regularizar (No sigue la propiedad de 1-Lipschitz).
- R: matriz en formato uint8 qué representa el residuo resultante después de calcular la quasidistancia.

3.2.1.2. Código

```
function [Q,R] = QErosionBased(I)
%QErosionBased Computation of Q as presented by S. Beucher
%   Computation of the quasidistance transformation image without
%   regularization Q and the residue image R using the algorithm
%   described by S. Beucher.
%     Input I has to be a uint8 matrix representing the image in
grayscale
%   Output Q is a uint8 matrix representing the quasidistance
%   transformation image without regularization.
%   Output R is a uint8 matrix representing the residue image.
%   Implementation by Eloi Sevilla.
```

Primero inicializamos algunas variables.

- SE: elemento estructurador morfológico qué utilizaran las erosiones de la imagen.
- pIx, pIy: dimensiones de la imagen
- W1, W2: resultados intermedios. Inicializamos W1 al valor de la imagen.
- i: número de iteración actual
- condition1: condición necesaria para continuar haciendo iteraciones dentro del bucle. Queremos ejecutar el bucle hasta qué volver a erosionar la imagen no haga nada, así que la condición es la siguiente: $condition1 = \neg W1 == W2$.

```
SE = strel('square', 3);
[pIx, pIy, ~]= size(I);

W1 = I;
Q = uint8(zeros(pIx, pIy));
R = uint8(zeros(pIx, pIy));
i = 0;
condition1 = true;
```

Seguidamente hacemos el bucle. En cada iteración el bucle hace las siguientes operaciones:

- Incrementamos el valor de i
- Calculamos la erosión de W1 y la guardamos en W2
- Calculamos W1-W2 y lo guardamos en residue
- Calculamos E, una matriz de booleanos que por cada pixel, contiene un valor true únicamente si el valor del residuo no es 0 y es mayor o igual que el valor que tenemos almacenado en R.
- Actualizamos los valores de Q. Cada píxel en qué E es true, es actualizado para qué sea igual al número de la iteración.
- Actualizamos los valores de R a los de residue en los pixeles en qué E es true. Podemos conseguir éste resultado calculando el máximo entre R y residue.
- Actualizamos W1 para qué sea igual qué W2, para qué la siguiente iteración se haga sobre el resultado parcial obtenida en ésta.
- Finalmente comprobamos si se cumple la condición de *condition1*. Podemos lograr esto calculando la suma de todos los elementos de residue, ya que $W1 == W2 \Rightarrow residue == 0$

```
    while condition1
       i = i+1;
       W2 = imerode(W1, SE);
       residue = W1-W2;
       E = (residue >= R & residue ~= 0);
       Q(E) = i;
       R = max(R, residue);
       W1=W2;
       condition1 = sum(sum(residue))~=0;
    end
end
```

3.2.1.3. Resultados

I	Q	R
		

Fig 12: Resultados de ejecutar QErosionBased. Fuente: Elaboración propia

3.2.2. RQErosionBased

3.2.2.1. Definición

Esta función regulariza Q para que siga la propiedad de 1-Lipschitz, utilizando el algoritmo Erosion-Based.
Entrada:

- Q: matriz en formato uint8 qué representa la quasidistancia antes de la regularización.

Salida:

- RQ: matriz en formato uint8 qué representa la quasidistancia después de la regularización.

3.2.2.2. Código

```
function [RQ] = RQErosionBased(Q)
%RQErosionBased Computation of RQ as presented by S. Beucher
%    Regularization of the quasidistance transformation image so it
follows
%   the 1-Lipschitz property as described by S. Beucher.
%   Input Q has to be a uint8 matrix representing the quasidistance
%   transformation image without regularization.
%   Output RQ is a uint8 matrix representing the quasidistance
%   transformation after regularization.
%   Implementation by Eloi Sevilla.
```

Primero inicializamos algunas variables.

- SE: elemento estructurador morfológico qué utilizaran las erosiones de la imagen.
- RQ: Imagen de la quasidistancia después de la regularización. La inicializamos para que empiece con el mismo valor que Q.
- condition2: condición para que se sigan haciendo iteraciones. Será cierto hasta que todos los píxeles de la imagen sigan la propiedad de 1-Lipschitz.

```
SE = strel('square', 3);
RQ = Q;
condition2 = true;
```

A continuación hacemos el bucle. En cada iteración el bucle hace las siguientes operaciones:

- Calculamos la erosión de RQ y la almacenamos en eRQ
- Calculamos la diferencia de RQ - eRQ y la almacenamos en W.
- Los píxeles en los que el valor de W es mayor que 1 son píxeles que no siguen la propiedad de 1-Lipschitz. Utilizamos la matriz de booleanos E para guardar esta información.
- Actualizamos los valores de RQ cuyas coordenadas correspondan a un valor true en E con el valor de las mismas coordenadas en la imagen erosionada + 1. Esto hace que el nuevo valor sea igual al del mínimo vecino +1, por lo que sí que cumple la propiedad de 1-Lipschitz.

- Finalmente recalculamos condition2. Solo queremos que sea false cuando todos los puntos cumplan la propiedad de 1-Lipschitz. Lo podemos calcular sumando todos los elementos de E. Si todos los puntos cumplen la propiedad, la suma será 0.

```
    while condition2
        eRQ = imerode(RQ, SE);
        W = RQ - eRQ;
        E = (W > 1);
        RQ (E) = eRQ(E) + 1;
        condition2 = sum(sum(E))~=0;
    end
end
```

3.2.2.3. Resultados

Q	RQ
	

Fig 13: Resultado de ejecutar RQErosionBased. Fuente: Elaboración propia

3.2.3. QQueueBased

3.2.3.1. Definición

Esta función computa la transformación de la quasidistancia de una imagen en escala de grises utilizando el algoritmo Queue-Based.
Entrada:

- I: matriz en formato uint8 qué representa la imagen original en escala de grises.

Salida:

- Q: matriz en formato uint8 qué representa la transformada de la quasidistancia antes de regularizar (No sigue la propiedad de 1-Lipschitz).
- R: matriz en formato uint8 qué representa el residuo resultante después de calcular la quasidistancia.

3.2.3.2. Código

```
function [Q,R] = QQueueBased(I)
%QQueueBased Computation of Q as presented by Raffi Enficiaud.
%   Computation of the quasidistance transformation image without
%   regularization Q and the residue image R using the algorithm
%   described by Raffi Enficiaud.
%     Input I has to be a uint8 matrix representing the image in
grayscale
%   Output Q is a uint8 matrix representing the quasidistance
%   transformation image without regularization.
%   Output R is a uint8 matrix representing the residue image.
%   Implementation by Eloi Sevilla.

    %% 2.1.1 Initialization
```

Primero inicializamos algunas variables.

- pIx, pIy: dimensiones de la imagen
- W1, W2: resultados intermedios. Los inicializamos al valor de la imagen.
- C: matriz de valores booleanos que indica qué píxeles tenemos que procesar.
- f1, f2: matrices que actúan cómo colas y contienen las coordenadas y valores de los pixeles a procesar en cada iteración del bucle.
- Ncoord: matriz auxiliar para facilitar el acceso a los píxeles vecinos a otro.
- index: variable auxiliar utilizada para saber cuántos elementos hay dentro de las colas.

```
    [pIx, pIy, ~]= size(I);
    W1 = I;
    W2 = I;
    Q = uint8(zeros(pIx, pIy));
    R = uint8(zeros(pIx, pIy));
```

```
    C = uint8(zeros(pIx, pIy));
    f1 = uint16(zeros(3, pIx*pIy));
    f2 = uint16(zeros(3, pIx*pIy));
    Ncoord = [-1 -1 -1 0 0 1 1 1; -1 0 1 -1 1 -1 0 1];

    index = 1;
```

En el primer bucle exploramos todos los píxeles p de la imágen:

```
    for x = 1:pIx
        for y = 1:pIy
            p = W1(x,y);
```

Para cada píxel, comprobamos los vecinos np al píxel:

```
            for n=Ncoord
                xn = x+n(1);
                yn = y+n(2);
                    if(xn > 0 && xn <= pIx && yn > 0 && yn <= pIy) % 
neighbour exists
                    np = W1(xn,yn);
                    if (np < p) % the point must be added to the list
```

Si el valor del vecino es menor al píxel, añadimos los datos de p a f1, y incrementamos el valor de index para indicar qué tenemos un valor más en f1.

```
                        newCol = [x;y;uint16(p)];
                        f1(:,index) = newCol;
                        index = index + 1;
                        break;
                    end
                end
            end
        end
    end
    %% 2.1.2 Propagation loop
```

Hacemos un segundo bucle, que irá haciendo iteraciones hasta que la cola f1 esté vacía. Para comprobar ésto inicializamos f1Elems para que contenga el número de elementos en f1. Inicializamos i (número de iteración) a 1.

```
    f1Elems = index-1;
    i = 1;
    while (f1Elems ~= 0)
    %%%% 2.1.2.1 computation of erosion of I
```

Exploramos todos los elementos de f1, por cada uno de estos elementos, marcamos que el pixel ya no está en la cola, igualando el valor de C correspondiente a sus coordenadas a 0.

```
        for index = 1:f1Elems
            newCol = f1(:,index);
            x = newCol(1);
            y = newCol(2);
            value = newCol(3);
            C(x,y) = 0;
```

Seguidamente, igualamos el valor del elemento que estemos procesando al menor valor de sus vecinos.

```
            for n=Ncoord
                xn = x+n(1);
                yn = y+n(2);
                    if(xn > 0 && xn <= pIx && yn > 0 && yn <= pIy) % neighbour exists
                     np = W1(xn,yn);
                     if(np<value)
                         value = np;
                     end
                end
            End
```

Finalmente, para el elemento que estemos procesando, actualizamos el valor en W2 para que sea igual al nuevo valor del elemento, y calculamos residue restando el valor anterior (almacenado en W1) menos el valor actual.

```
            f1(:,index) = [x;y;uint16(value)];
            W2(x,y) = value;
            residue = W1(x,y) - value;
```

Si el valor de residue es mayor o igual que el almacenado en R, actualizamos el valor en R a residue y el valor de Q a i.

```
            if (residue >= R(x,y)
                %%%% 2.1.2.2 update of the distance Q and residue R
                R(x,y) = residue;
                Q(x,y) = i;
            end
        end
    %%%% 2.1.2.3 continue loop
```

En el siguiente bucle, volvemos a iterar entre todos los valores de f1, y actualizamos los valores de W1 al nuevo valor.

```
        index2 = 1;
        for index = 1:f1Elems
            newCol = f1(:,index);
            x = newCol(1);
            y = newCol(2);
            value = newCol(3);
            W1(x,y) = W2(x,y);
```

Una vez hecho esto, exploramos todos sus vecinos y si nos encontramos con un vecino con un valor mayor al del píxel actual, y que aún no esté en la cola (indicado porque su valor en C será 0), metemos el píxel vecino en la cola f2 y actualizamos su valor en C a 1 para indicar qué lo hemos metido en la cola. Utilizamos la variable index2 para llevar la cuenta de elementos en f2.

```
            for n=Ncoord
                xn = x+n(1);
                yn = y+n(2);
                    if(xn > 0 && xn <= pIx && yn > 0 && yn <= pIy) % 
neighbour exists
                    np = uint16(W2(xn,yn));
                    if(np>value && C(xn,yn) == 0)
                        newColumn = [xn;yn;uint16(np)];
                        f2(:,index2) = newColumn;
                        index2 = index2 + 1;
                        C(xn,yn) = 1;
                    end
                end
            end
        End
```

Finalmente, traspasamos los valores de f2 a f1, actualizamos el valor de f1Elems para que refleje el nuevo número de elementos e incrementamos el valor de i en 1.

```
        f1Elems = index2-1;
        for index = 1:f1Elems
            f1(:,index) = f2(:,index); % f1 = f2
            f2(:,index) = uint16([0;0;0]); % f2 = 0
        end
        i = i+1;
    end
end
```

3.2.3.3. Resultados

I	Q	R
		

Fig 14: Resultado de ejecutar QQueueBased. Fuente: Elaboración propia

3.2.4. RQQueueBased

3.2.4.1. Definición

Esta función regulariza Q para que siga la propiedad de 1-Lipschitz, utilizando el algoritmo Queue-Based.

Entrada:

- Q: matriz en formato uint8 que representa la quasidistancia antes de la regularización.

Salida:

- RQ: matriz en formato uint8 que representa la quasidistancia después de la regularización.

3.2.4.2. Código

```
function [RQ] = RQQueueBased(Q)
%RQQueueBased Computation of RQ as presented by Raffi Enficiaud.
%    Regularization of the quasidistance transformation image so it
follows
%   the 1-Lipschitz property as described by Raffi Enficiaud.
%   Input Q has to be a uint8 matrix representing the quasidistance
%   transformation image without regularization.
%   Output RQ is a uint8 matrix representing the quasidistance
%   transformation after regularization.
%   Implementation by Eloi Sevilla.
```

Empezamos inicializamos algunas variables.

- pIx, pIy: dimensiones de la imagen
- C: matriz de valores booleanos que indica qué píxeles tenemos que procesar.
- Ncoord: matriz auxiliar para facilitar el acceso a los píxeles vecinos a otro.
- elems y hq: variables utilizadas para hacer la función de hierarchical queue, explicado en el anexo

```
[pIx, pIy, ~]= size(Q);
Ncoord = [-1 -1 -1 0 0 1 1 1; -1 0 1 -1 1 -1 0 1];
RQ = Q;
C = uint8(zeros(pIx, pIy));
elems = zeros(256,1);
hq = zeros(3, pIx*pIy,256);
```

En este primer bucle exploramos todos los píxeles de la imágen.

```
%% first loop
for x = 1:pIx
    for y = 1:pIy
        p = RQ(x,y);
```

Para cada píxel, observamos sus vecinos, y si encontramos un vecino que no siga la propiedad de 1-Lipschitz, indicamos que hay que procesarlo igualando el valor en C con sus coordenadas a 1.

```
            for n=Ncoord
                xn = x+n(1);
                yn = y+n(2);
                    if(xn > 0 && xn <= pIx && yn > 0 && yn <= pIy) %
neighbour exists
                    v = RQ(xn,yn);
                    if(v>p+1)
                        C(xn,yn) = 1;
                    end
                end
            end
        end
    end
```

Hacemos un segundo bucle, en el que solamente procesamos los píxeles con valor 1 en C.

```
    %% second loop
    dirty = 0;
    for x = 1:pIx
        for y = 1:pIy
            p = RQ(x,y);
            if (C(x,y)==1)
                for n=Ncoord
```

Para cada uno de éstos píxeles, cambiamos su valor para que sea igual al valor del vecino v que no esté marcado para procesar (C(v) == 0) con menor valor. Igualamos dirty a 1 y actualizamos la variable leastNeighborTrue para actualizar la hierarchical queue y RQ más adelante, fuera del bucle.

```
                    xn = x+n(1);
                    yn = y+n(2);
                     if(xn > 0 && xn <= pIx && yn > 0 && yn <= pIy) %
neighbour exists
                        leastNeighbor = Q(xn, yn) +1;
                        if (C(xn,yn)==0 && p>leastNeighbor)
                            p = leastNeighbor;
                            leastNeighborTrue = leastNeighbor;
                            dirty = 1;
                        end
                    end
                end
```

```
                if(dirty==1)
                    % hq = hq + p at priority leastNeighbor
                    position = elems(leastNeighborTrue) + 1;
                    elems(leastNeighborTrue) = position;
                                hq(:,position,leastNeighborTrue)  =
[x;y;uint16(p)];
                    % actualizar RQ al final
                    RQ(x,y) = p;
                    dirty = 0;
                end
            end
        end
    end
```

Por último hacemos un tercer bucle, que irá haciendo iteraciones hasta que la hierarchical queue esté vacía.

```
    %% third loop
    while(sum(elems)~=0)
```

Primero buscamos la cola con máxima prioridad que no esté vacía, y almacenamos su prioridad en pr y el número de elementos que tiene en prSize. La cola con máxima prioridad en este caso es la cola con menor valor numérico, ya que el algoritmo va disminuyendo los valores de los píxeles para que sigan la propiedad 1-Lipschitz, así que empezando por los puntos con menos valor se evita la necesidad de computar el mismo píxel varias veces que se da cuando se disminuye el valor de un píxel vecino a un nuevo valor por debajo de la corrección. En el anexo 2 se puede ver una explicación gráfica de la importancia de este orden.

```
        for i = (1:256)
            pr = i; % pr = highest priority of hq
            if(elems(pr) ~= 0)
                prSize = elems(pr);
                break;
            end
        end
```

Seguidamente, iteramos por todos los puntos en la cola con máxima prioridad, y miramos sus vecinos en busca de un vecino que no siga la propiedad de 1-Lipschitz.

```
        for p = hq(:,1:prSize,pr)
            x = p(1);
            y = p(2);
            value = p(3);
            for n=Ncoord
                xn = x+n(1);
                yn = y+n(2);
                    if(xn > 0 && xn <= pIx && yn > 0 && yn <= pIy) %
neighbour exists
                    if(RQ(xn,yn) > pr + 1)
```

Para todos los píxeles vecinos que no sigan la propiedad, los insertamos en la hierarchical queue con prioridad pr+1, y actualizamos su valor en RQ también a pr + 1.

```
                        temp = elems(pr + 1) + 1;
                        elems(pr +1) = temp;
                        hq(:,temp,pr+1) = [xn;yn;uint16(pr+1)];
                        RQ(xn,yn) = pr + 1;
                    end
                end
            end
        end
```

Finalmente, vaciamos la cola con prioridad pr.

```
        elems(pr) = 0;
    end
end
```

3.2.4.3Resultados

Q	RQ
	

Fig 15: Resultado de ejecutar RQQueueBased. Fuente: Elaboración propia

4. Experimentos

A continuación se documentan varios experimentos que se han hecho para comprobar la eficiencia y la eficacia de los diferentes algoritmos.

4.1. Cálculo de tiempo

Utilizando las funciones de “tic” [9] y “toc” [10] de matlab podemos calcular y comparar el tiempo en segundos que tarda a ejecutarse las distintas funciones con varias imágenes:

	Tiempo QErosionBased	Tiempo RQErosionBased	Tiempo QQueueBased	Tiempo RQQueueBased
faded.png (270x270)	0.0783s	0.0020s	23.1442s	0.4729s
lenna.tif (512x512)	0.4632s	0.1200s	75.8611s	6.1777s
lake.png (512x512)	0.8614s	0.1311s	84.9643s	4.7354s
cameraman.png (512x512)	0.9573s	0.3730s	72.1445s	6.1546s

Fig 16: Comparación de tiempos de ejecución. Fuente: Elaboración propia

Con estos tiempos podemos apreciar que el tiempo de ejecución varía bastante dependiendo de las propiedades de la imagen procesada, principalmente en base del tamaño, pero se mantiene constante que la ejecución de las versiones Erosion-Based son bastante más rápidas que la respectiva versión Queue-Based. Por este motivo, utilizaremos la versión Erosion-Based para siguientes experimentos que se centren más en la eficacia del algoritmo, ya que los resultados son exactamente iguales. También cabe destacar que el tiempo de las ejecuciones varían bastante dependiendo de las condiciones del ordenador, así que cada imagen se ha usado cómo entrada varias veces, y el tiempo anotado es la media de las ejecuciones.

4.2. Imagen degradada

Para comprobar la eficacia de la técnica de la quasidistancia, intentaremos segmentar la imagen faded3.png utilizando distintas alternativas. Pese a verse a simple vista que la imagen contiene 2 círculos sobre un fondo oscuro, el degradado de los círculos y el fondo hace que automatizar el proceso de segmentación no sea trivial, cómo veremos en los resultados de este experimento.

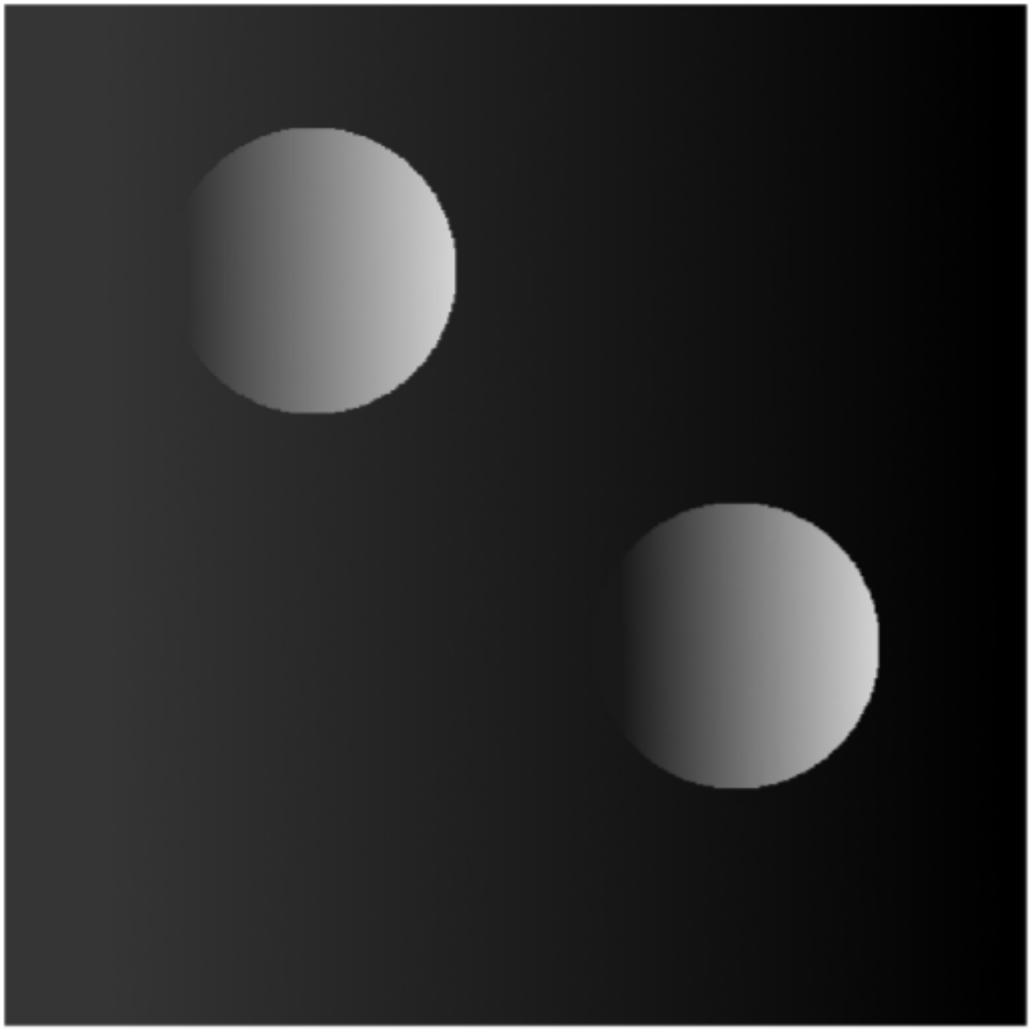


Fig. 17:Imagen faded3.png, creada artificialmente para resaltar dificultades de los algoritmos de segmentación. Tanto los círculos como el fondo tienen un degradado de color, y hay un momento en el que el color de los círculos tienen el mismo color que el fondo, lo que provoca que el gradiente de la imagen no termine de cerrarse. Fuente: Elaboración propia

4.2.1. Segmentación usando watershed sobre la quasidistancia

Aplicamos watershed sobre la inversa de la quasidistancia calculada con los algoritmos implementados en este proyecto. La segmentación resultante es esta:

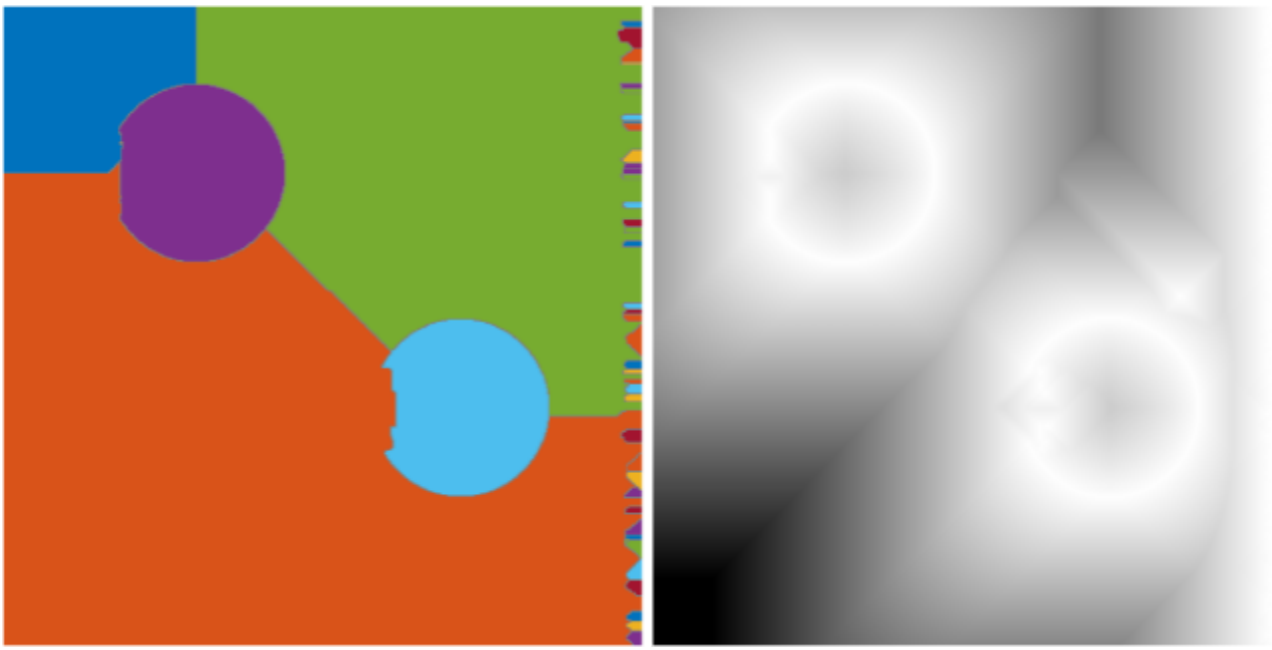


Fig 18: Izquierda: Resultado de segmentar faded3.png usando la Qdist. Derecha: Resultado de la Qdist. Fuente: Elaboración propia

Pese a no ser un resultado perfecto, ha segmentado los dos círculos correctamente. Hay sobre-segmentación en el fondo debido a la forma creada por la Qdist, pero con un poco de postprocesado se podría llegar a diferenciar entre la segmentación correspondiente a objetos reales y la sobre-segmentación causada por error.

La característica de la imagen de la Qdist es el hecho de que se forma una especie de “volcanes” alrededor de los dos círculos. El watershed da los resultados esperados dentro de los volcanes, ya que inicia en el centro de estos y se van rellenando, pero en el fondo el watershed empieza por las tres esquinas más oscuras, por eso produce sobre segmentación y crea unas fronteras adicionales en las zonas donde no se llega al mismo mínimo que en las esquinas.

4.2.2. Segmentación usando watershed directamente sobre la inversa de la imagen

Aplicamos watershed sobre la inversa de la imagen original. Esta técnica es muy simple, pero los resultados dejan mucho que desear. Podemos observar que hay muchísima sobre-segmentación, y mientras que en el resultado se puede apreciar que labels pertenecen a los círculos, no es buena idea usar este método.

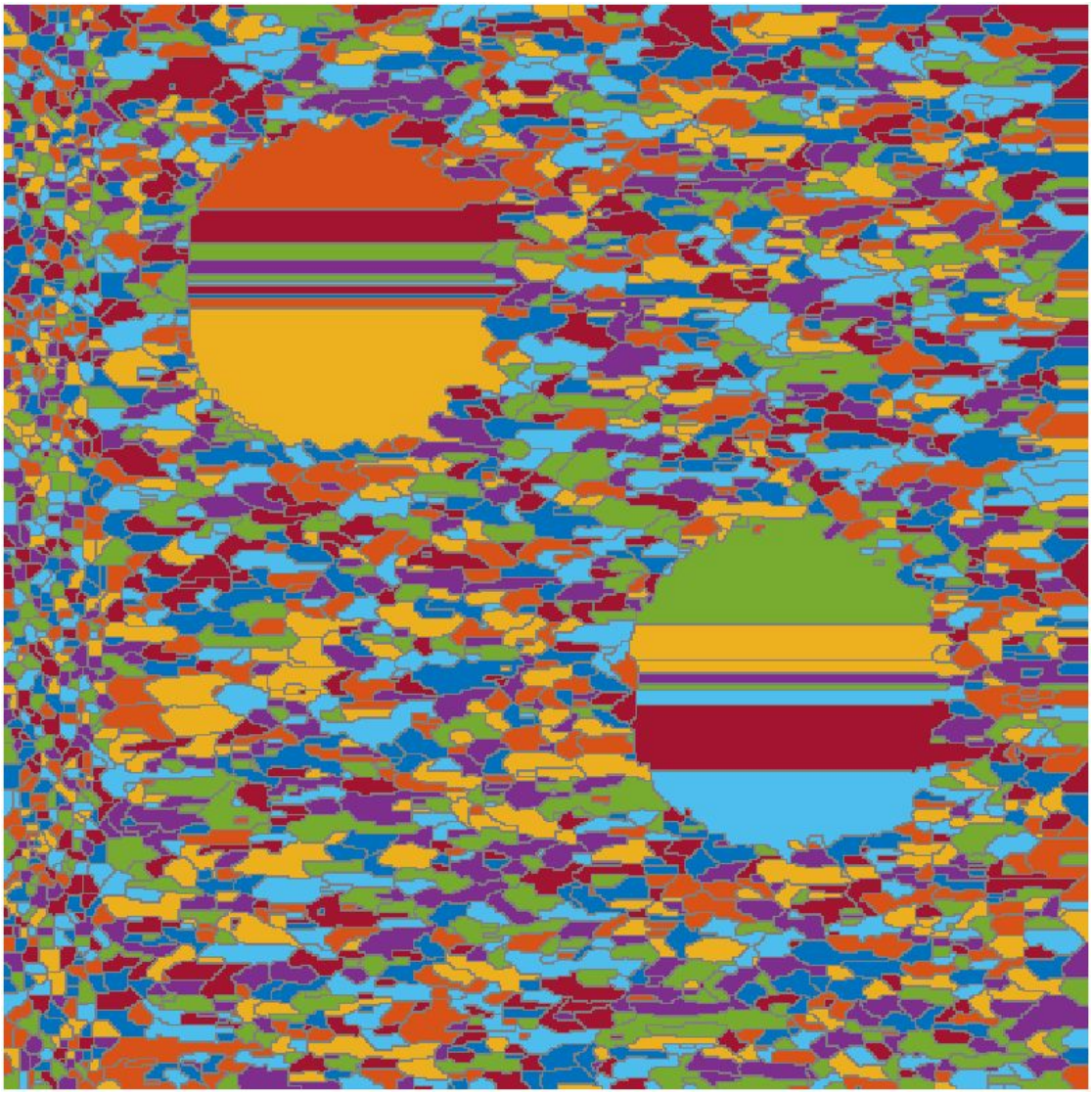


Fig 19: Resultado de segmentar faded3.png usando la inversa de la imágen.
Fuente: Elaboración propia

4.2.3. Segmentación usando watershed sobre la transformada de la distancia tradicional

El problema de este método es la necesidad de binarizar la imagen previamente para poder utilizar el algoritmo de la transformada de la distancia.En este experimento he binarizado la imagen utilizando la media de los valores de todos los píxeles como límite. Podría haber utilizado otro método de binarización que funcionase mejor para esta imágen, pero quería recalcar la principal problemática de este método: conseguir binarizar la imagen correctamente. El resultado de la segmentación no es de ninguna utilidad, debido a la mala binarización.

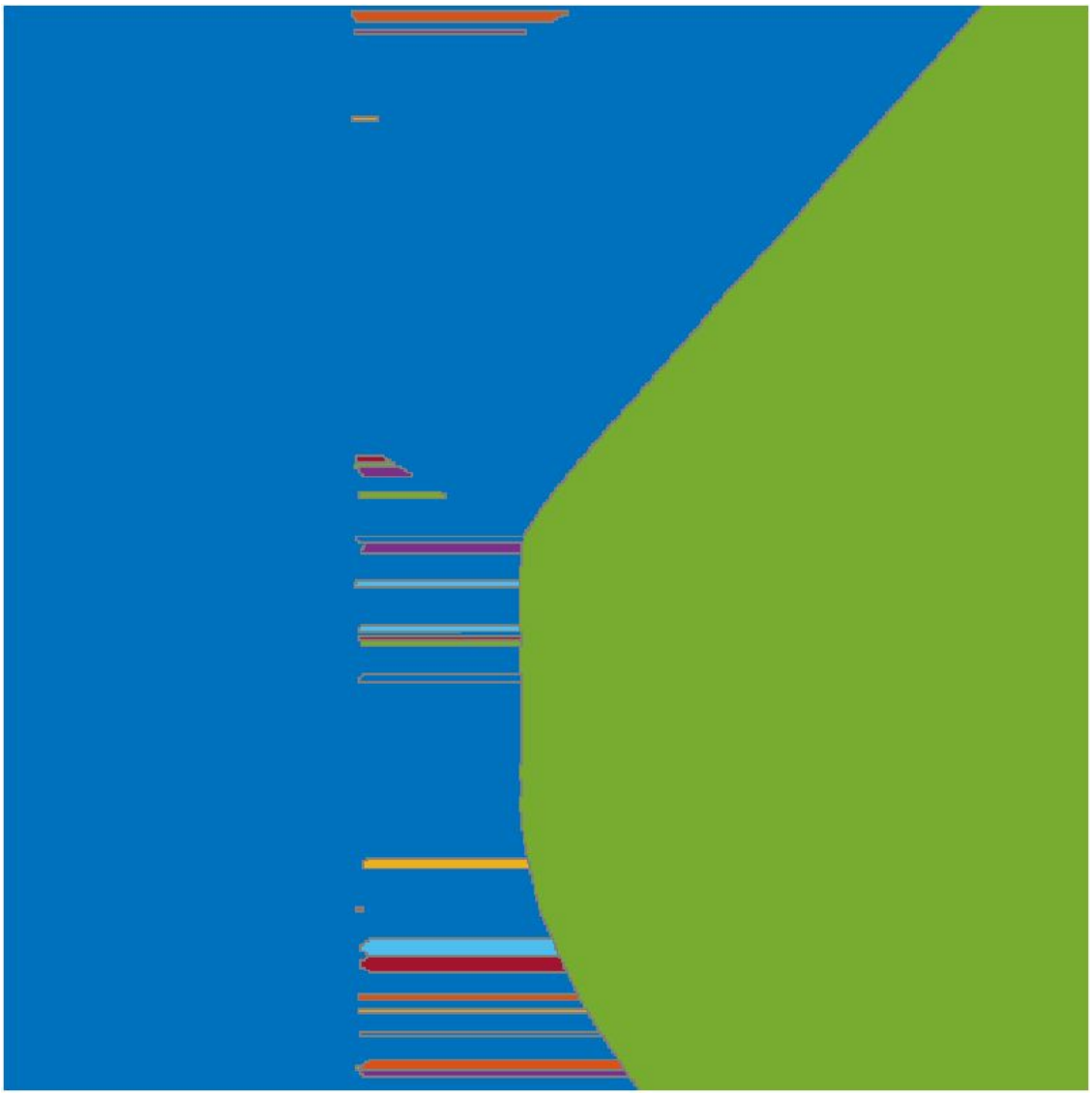


Fig 20: Resultado de segmentar faded3.png usando la Tdist. Fuente: Elaboración propia

4.2.4. Segmentación usando watershed sobre la imagen del gradiente

En este método, se aplica el watershed sobre la imagen del gradiente. El resultado está demasiado sobre-segmentado cómo para tener alguna utilidad. Aunque se puede percibir dónde están los círculos, el resultado sigue teniendo una calidad un poco pobre

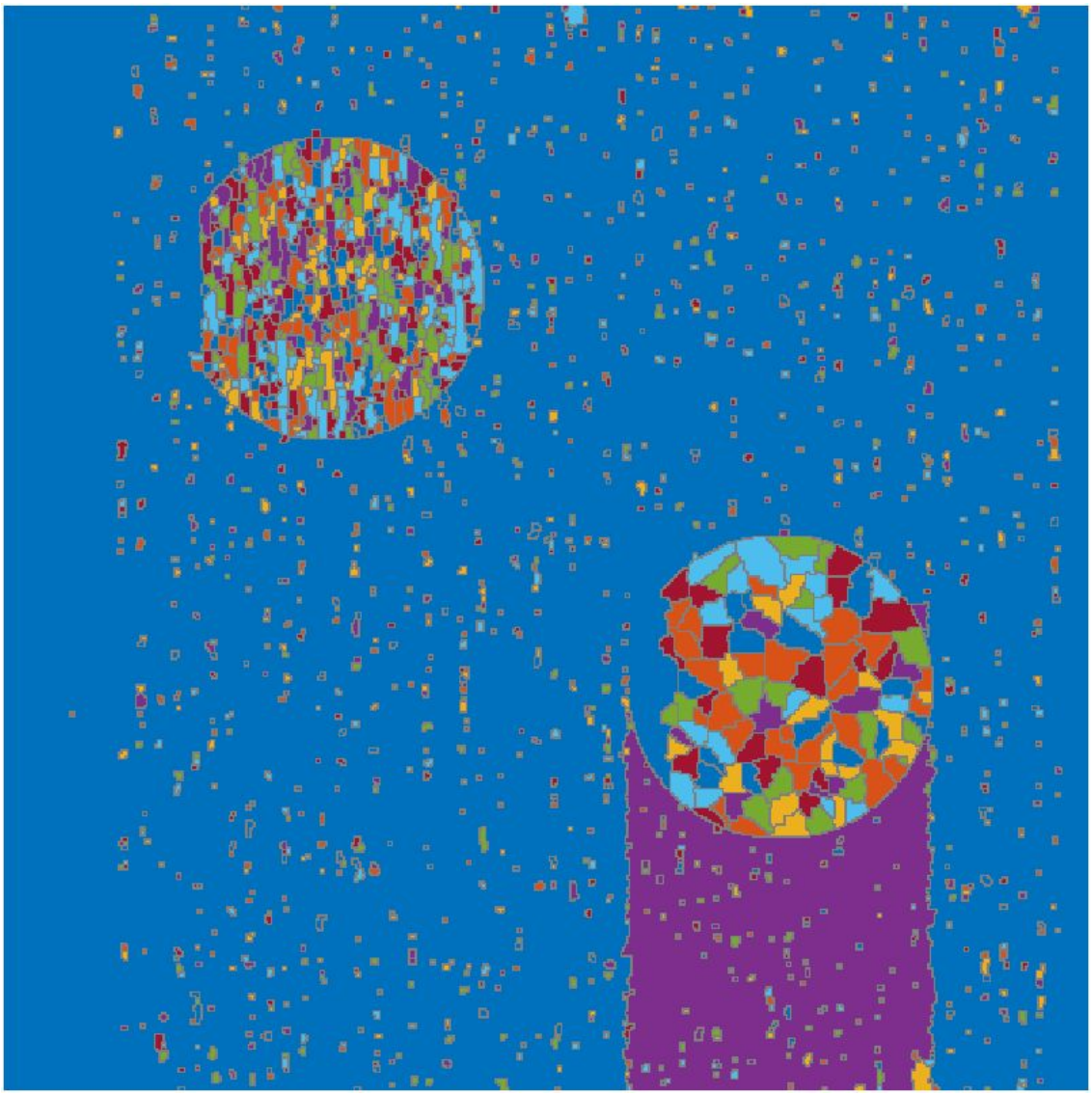


Fig 21: Resultado de segmentar faded3.png usando el gradiente. Fuente: Elaboración propia

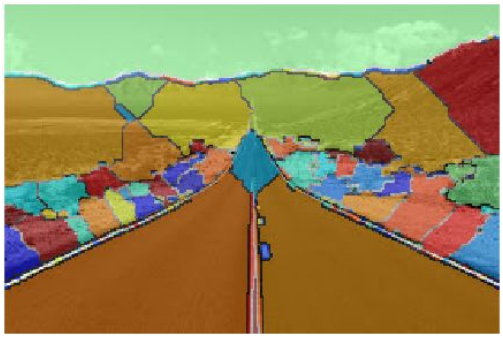
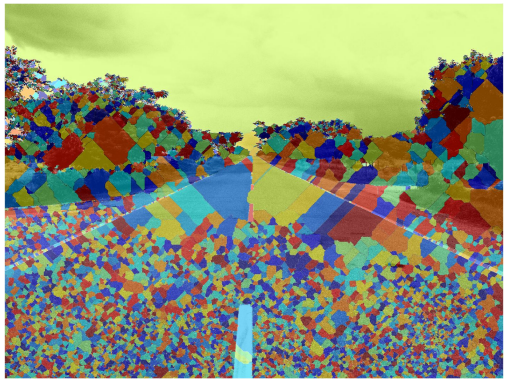
4.2.5. Comparación

Pese a no ser perfectos, los resultados después de utilizar la quasidistancia son bastante mejores que los obtenidos utilizando alternativas similares, así que este algoritmo puede resultar muy útil en casos reales de segmentación. Podrían mejorarse los resultados de todos los métodos para disminuir la sobre-segmentación, pero el objetivo de este experimento es principalmente comparar los resultados que proporcionan utilizar las técnicas directamente.

4.3. Segmentación de carretera

En este experimento probamos el algoritmo en un caso real en el que podría ser interesante utilizarlo. Intentaremos segmentar varias imágenes de carreteras, para comprobar si los resultados podrían ser utilizables en un sistema de conducción autónoma. Utilizaremos distintas imágenes de carreteras para comparar los resultados bajo distintas condiciones.
Para mejorar los resultados se utilizarán algunas técnicas comunes en visión por computador. Para facilitar la segmentación de tanto las partes claras y las partes oscuras, en lugar de utilizar el algoritmo sobre la imagen original, primero se calcula el gradiente de la imagen y se aplica el algoritmo sobre el gradiente. Haciendo esto se logra que la segmentación se haga sobre la diferencia de color, es decir, se segmentan por igual los objetos claros y los objetos oscuros. De esta manera la segmentación resultante es más correcta. Otro sistema utilizado para mejorar los resultados es filtrar los mínimos locales de la imagen gradiente. El filtro elimina los mínimos locales con menor profundidad a la profundidad deseada. De esta manera hay menos puntos en los que se inicia el watershed y por lo tanto hay menos sobre-segmentación. Los resultados después de este proceso son los siguientes.

4.3.1. Resultados

Num imagen	Imagen Original	Segmentación
1		
2		

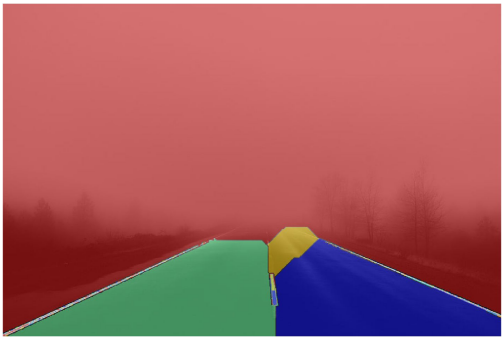
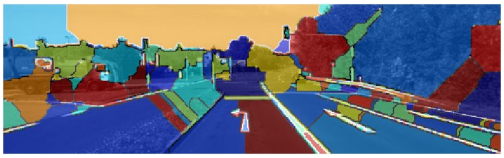
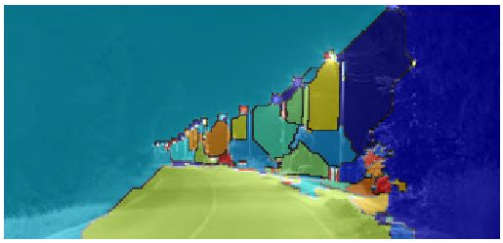
3		
4		
5		

Fig 22: Resultados de segmentar varias imágenes de carreteras usando la Qdist.
Fuente: Elaboración propia

4.3.2. Observaciones

Los resultados de este experimento son bastante variados, pero se mantiene constante qué hay sobre-segmentación, y aunque la segmentación resultante no se puede utilizar directamente para encontrar la carretera, podría utilizarse en combinación a otros sistemas para diferenciar las líneas pintadas en el asfalto.
Un caso con especialmente mal resultado es el de la segunda imágen. Esto se debe principalmente por los fragmentos blancos que se ven en la carretera y por la excesiva resolución de la imágen. Podemos mejorar bastante el resultado si bajamos un poco la resolución de la imágen y emborronamos la imagen un poco para difuminar un poco los puntos blancos. El resultado en este caso es el siguiente, mucho mejor:

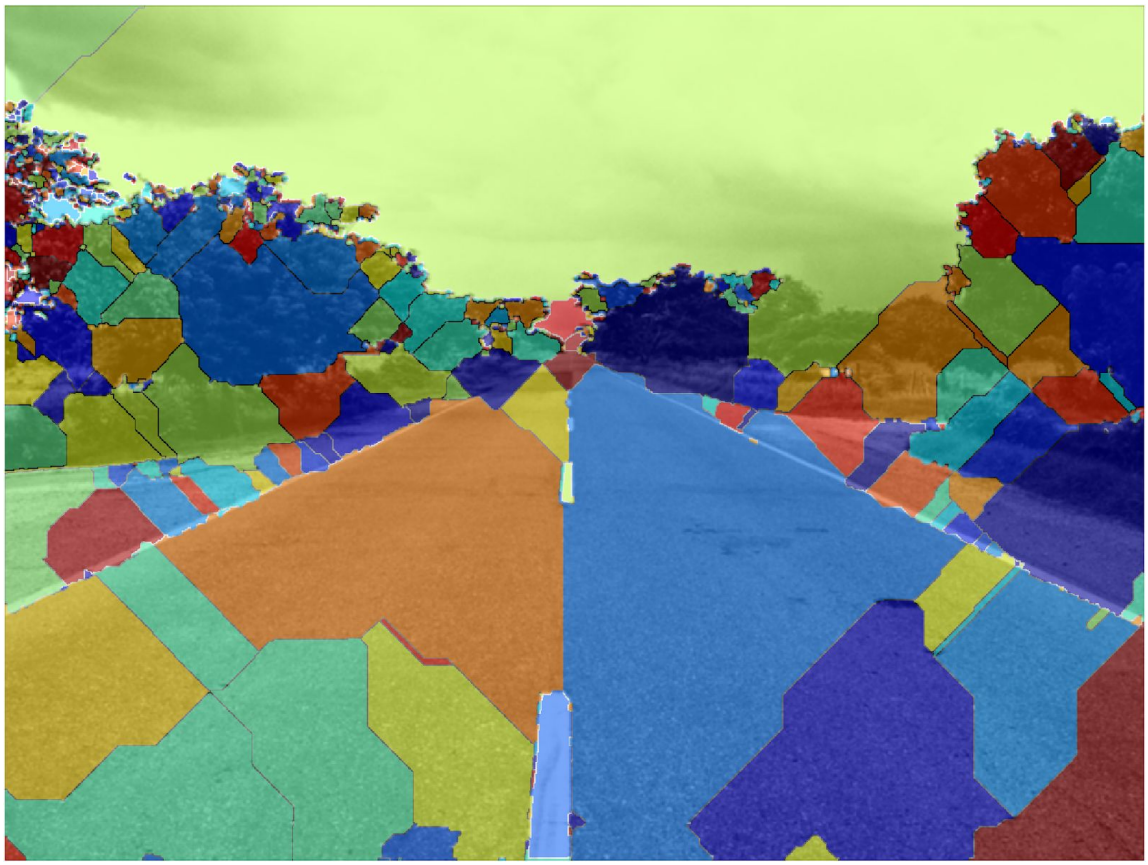


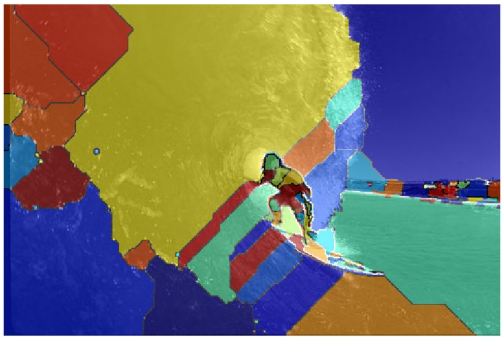
Fig 23: Resultados de segmentar la imágenes de carretera 2 usando la Qdist preprocesando la imagen. Fuente: Elaboración propia

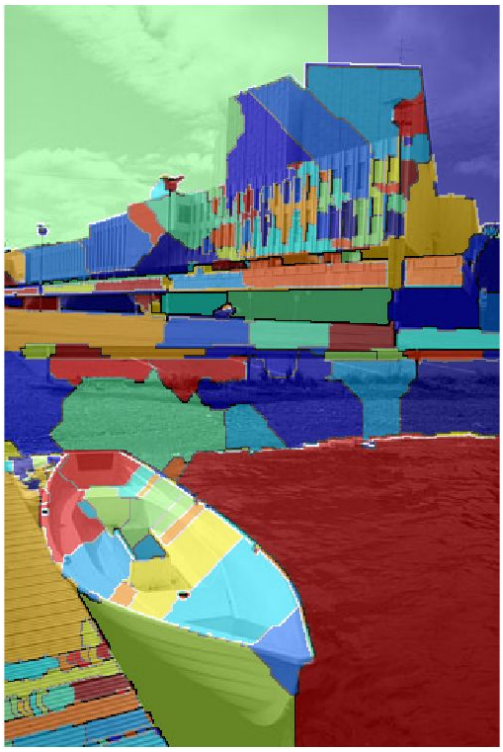
Estos resultados son bastante decentes,pero no es suficiente para hacer el reconocimiento de carreteras de un sistema de conducción autónoma, ya que es demasiado dependiente de las condiciones en las que se encuentra la carretera, las condiciones metereológicas y la iluminación. No obstante podría servir de soporte junto otros sistemas. Ya que detecta la pintura blanca sobre el asfalto bastante bien, sobretodo cuando las líneas son grandes y están cerca de la cámara, una función para la que podría funcionar bastante bien és para asistir en el aparcamiento automático.

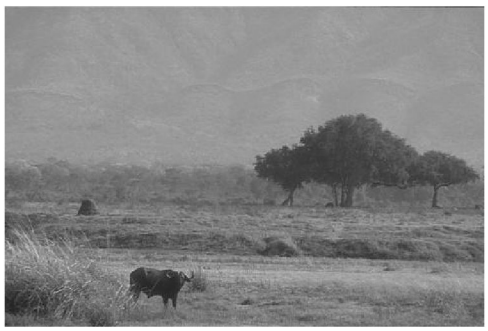
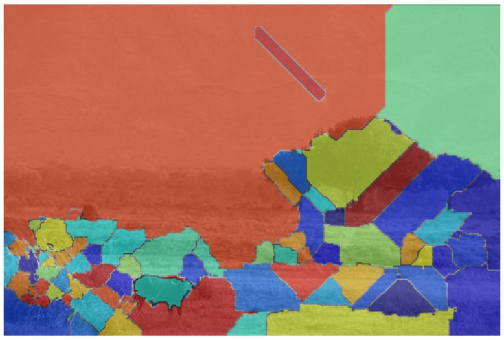
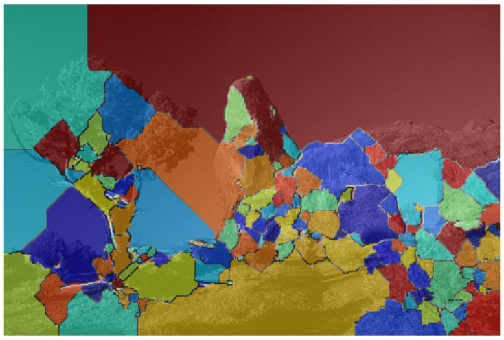
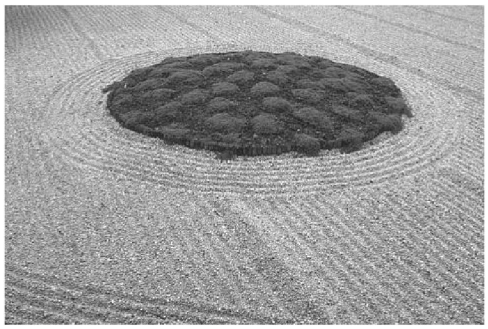
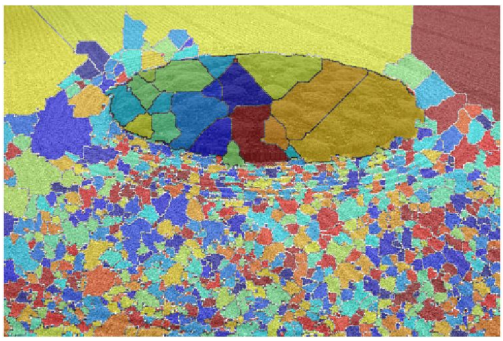
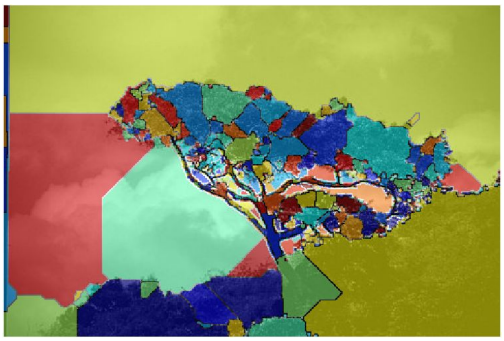
4.4. Segmentación de imágenes de test

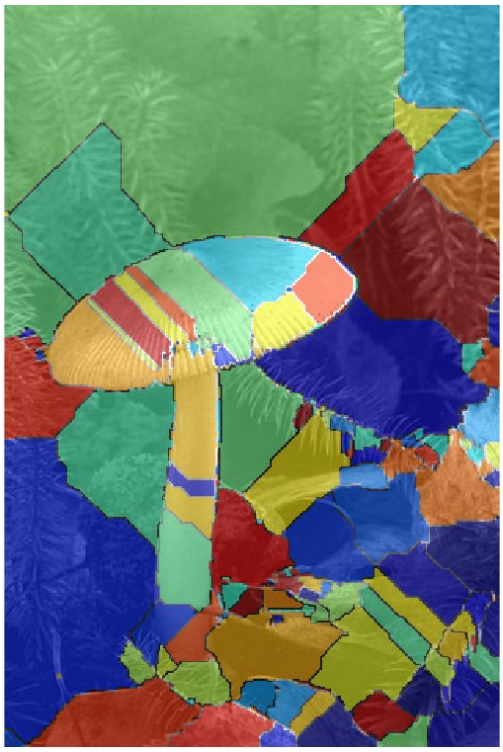
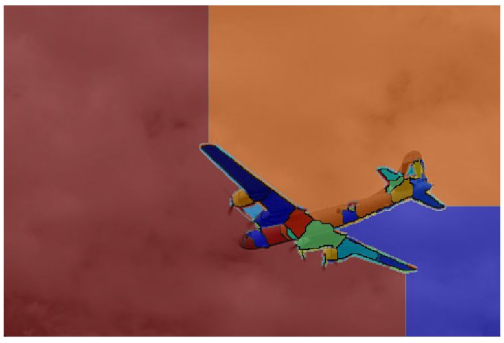
A continuación se analiza el resultado de aplicar el algoritmo a varias imágenes un poco más variadas, para observar los resultados sobre casos más complicados de segmentación. Una vez más, se aplica el algoritmo sobre el gradiente de la imagen y se filtran los mínimos locales para mejorar un poco la segmentación obtenida. Todas las imágenes utilizadas en este test se han sacado del dataset de Berkeley [11].

4.4.1. Resultados

Num imagen	Imagen Original	Segmentación
1		
2		
3		

4		
5		
6		

7		
8		
9		
10		

11		
12		
13		

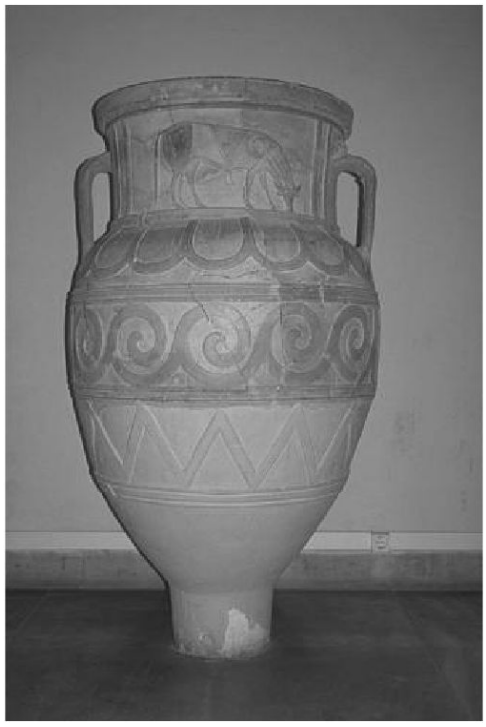
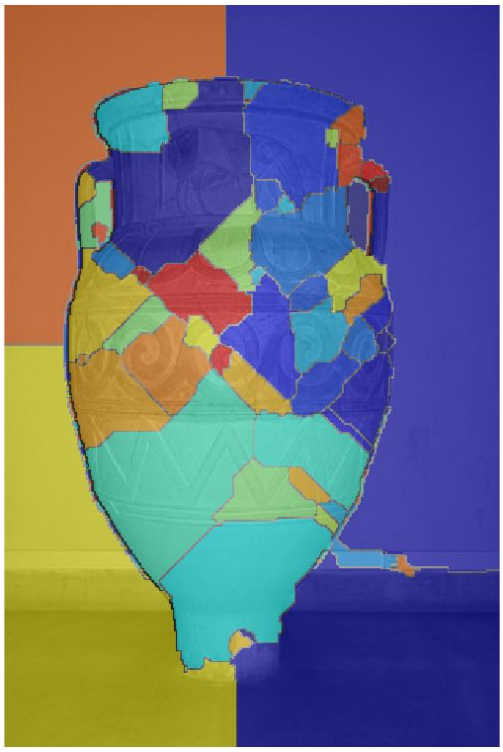
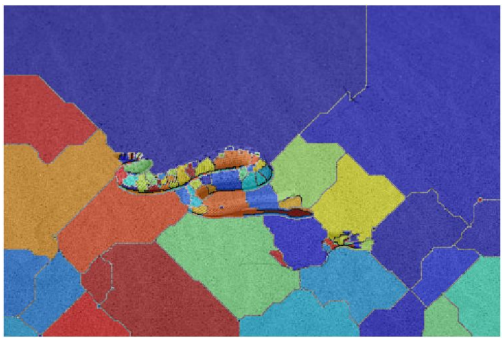
14		
15		

Fig 24: Resultados de segmentar varias imágenes usando la Qdist.
Fuente: Elaboración propia

4.4.2. Observaciones

En los resultados de este test se puede observar la calidad general de la segmentación resultante. Claramente hay bastante sobre-segmentación, sobretodo en algunas imágenes más complicadas como la 9 o la 11, sin embargo, este es un problema con el que era bastante razonable encontrarse. Así pues, pese a tener claras debilidades, los resultados de este algoritmo son bastante decentes teniendo en cuenta la simplicidad y velocidad del algoritmo.

5. Conclusiones

En conclusión, el algoritmo de la quasi-distancia proporciona resultados interesantes en el campo de la visión por computador que lo convierten en una herramienta a tener en cuenta cuando se necesita segmentar una imagen. Con los resultados obtenidos de los experimentos, podemos apreciar que el algoritmo tiene potencial para utilizarse de soporte en programas que requieran segmentar imágenes. Pese a no ser perfectos, los resultados se pueden mejorar aplicando otras técnicas adicionales, llegando a segmentaciones bastante prometedoras que podrían utilizarse por ejemplo en sistemas de aprendizaje autónomo. Personalmente, me parece que los resultados obtenidos sobre carreteras son excepcionalmente interesantes, y podrían utilizarse en paralelo a otros sistemas a modo de apoyo.

Cómo punto negativo, este algoritmo es demasiado dependiente de las condiciones de la imagen procesada, así que no sería suficiente como sistema único en un programa que requiriese exactitud en los resultados. Esto se debe a que el algoritmo depende principalmente en el color, así que es natural que pequeñas variaciones del color cambien los resultados.

Respecto a las dos versiones implementadas, resulta sorprendente que la implementación del algoritmo Erosion-Based termine siendo bastante más rápida que la Queue-Based. Esta diferencia de tiempos se debe posiblemente a la implementación en bajo nivel de las funciones utilizadas en Matlab. Mientras que entre las dos versiones de los algoritmos, la Queue-Based debería ser más eficiente, las funciones de Matlab están optimizadas para el procesamiento de imágenes, lo que hace que las erosiones utilizadas en la versión Erosion-Based terminen siendo menos funciones internamente que los múltiples accesos a matrizes de la versión Queue-Based.

6. Anexos

6.1. Anexo 1: Implementación Hierarchical queue

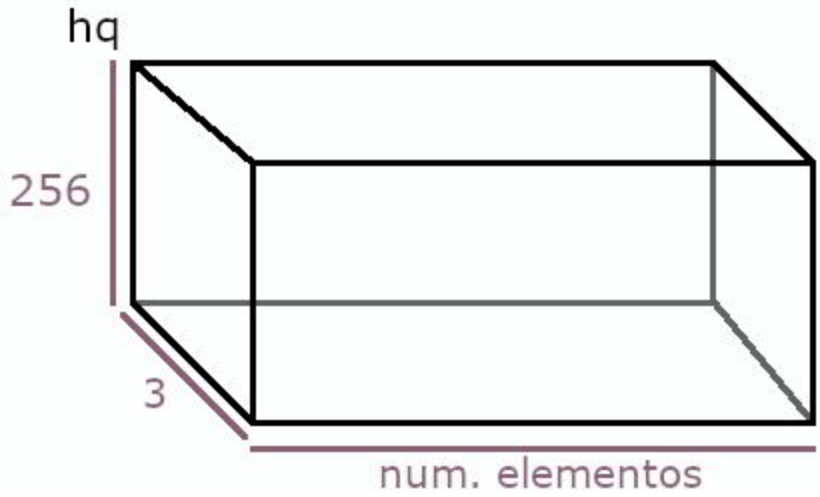


Fig 25: Descripción gráfica de la implementación de la Hierarchical queue.
Fuente: Elaboración propia

elems: contiene 256 elementos, cada uno representa el número de elementos correspondientes a una prioridad. Por ejemplo, si elems[5] = 7, hay 7 elementos con prioridad 5.
hq: matriz de 3 dimensiones:

1. Dimensión correspondiente al número de prioridades, de tamaño 256. Cada elemento representa una prioridad.
2. Dimensión correspondiente al número de elementos en la hierarchical queue, con tamaño variable dependiendo en el tamaño de la imagen procesada.
3. Dimensión que corresponde a los elementos que se guardan de cada píxel, de tamaño 3. Estos elementos son los siguientes:
    a. Coordenada x del píxel.
    b. Coordenada y del píxel.
    c. Valor del píxel.

6.2. Anexo 2: Orden de la regularización

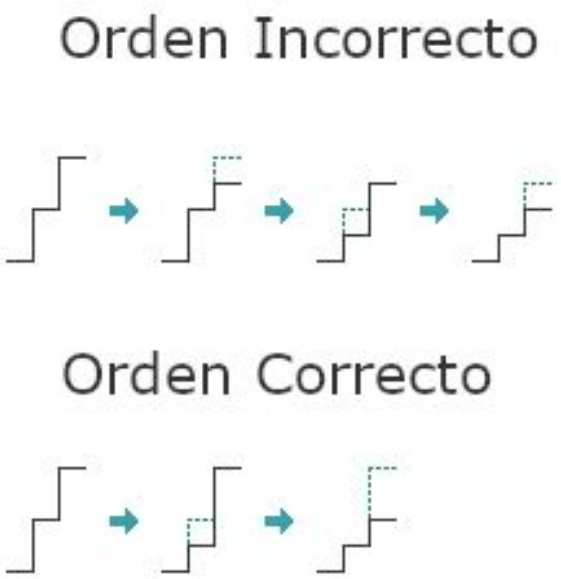


Fig 26: En este ejemplo a pequeña escala se aprecia la diferencia que causa el orden de la regularización. Si se empieza a regularizar por el mayor elemento, el valor de éste cambia dos veces. No obstante, si se empieza la regularización por el elemento de menor valor, solamente se cambia el valor de los elementos una vez, así que se llega al mismo resultado con una iteración menos. Fuente: Elaboración propia.

6.3. Anexo 3: Código del test de cálculo del tiempo (TimeTest.m)

```
ImgName = 'filename.png'; % image to load
J = imread(ImgName);
I = uint8(J);

%% Computations with Erosion-Based version
tic;
[Q1,R1] = QErosionBased(I);
QErosionBasedTime = toc;

tic;
RQ1 = RQErosionBased(Q1);
RQErosionBasedTime = toc;

%% Computations with Queue-Based version

tic;
[Q2,R2] = QQueueBased(I);
QQueueBasedTime = toc;

tic;
RQ2 = RQQueueBased(Q2);
RQQueueBasedTime = toc;

%% Display time
diff = RQ1 - RQ2;

imshow(diff, []);
disp(QErosionBasedTime);
disp(RQErosionBasedTime);
disp(QQueueBasedTime);
disp(RQQueueBasedTime);
```

6.4. Anexo 4: Código del test de Imagen degradada (FadedTest.m)

```
ImgName = 'faded3.png'; % image to load
J = imread(ImgName);
I = uint8(J);
%% Computations with Q-dist
[Q1,R1] = QErosionBased(I);
RQ1 = RQErosionBased(Q1);
D = -(double(RQ1)/255);
L1 = watershed(D);
rgb1 = label2rgb(L1,'lines',[.5 .5 .5]);
figure; imshow(rgb1);
%% Computations with watershed directly over the inverse
Inv = -double(I);
L2 = watershed(Inv);
rgb2 = label2rgb(L2,'lines',[.5 .5 .5]);
figure; imshow(rgb2);
%% Computations with Tdist (binarizing first)
AbsMean = mean(mean(I));
binI = I>AbsMean;
D2 = bwdist(binI);
L3 = watershed(D2);
rgb3 = label2rgb(L3,'lines',[.5 .5 .5]);
figure; imshow(rgb3);
%% Computations with gradient image
[Gx,Gy] = imgradientxy(I);
[Gmag,Gdir] = imgradient(Gx,Gy);
maxim = max(max(Gmag));
GmagNor = uint8(Gmag/maxim*255);
L4 = watershed(GmagNor);
rgb4 = label2rgb(L4,'lines',[.5 .5 .5]);
figure; imshow(rgb4);
```

6.5. Anexo 5: Código de los tests de Segmentación sobre fotografías (PhotoSegmentationTest.m)

```
ImgName = img.jpg'; % image to load
J = imread(ImgName);
I = uint8(J);
%% Computations with Q-dist

[Gmag,Gdir] = imgradient(I);
maxim = max(max(Gmag));
GmagNor = uint8(Gmag/maxim*255);

H = 50;
M = imhmin(GmagNor, H);

InvGmagNor = 255 - M;

[Q1,R1] = QErosionBased(InvGmagNor);

RQ1 = RQErosionBased(Q1);
D = -(double(RQ1)/255);


L1 = watershed(D);

figure; B1 = labeloverlay(I,L1); imshow(B1);
```

7. Referencias

[1] “Distance Transform” [en línia]. [Consulta: 20 febrero 2020]. Disponible en: <https://homepages.inf.ed.ac.uk/rbf/HIPR2/distance.htm>
[2] Raffi Enficiaud: Queue and Priority Queue Queue Based Algorithms for Computing the Quasi-distance Transform, 2010.
[3] “IMAGE SEGMENTATION AND MATHEMATICAL MORPHOLOGY” [en línia]. [Consulta: 22 febrero 2020]. Disponible en: <http://www.cmm.mines-paristech.fr/~beucher/wtshed.html>
[4] Joel W. Robbin: Continuity and Uniform Continuity, 2010.
[5] “Hierarchical Queues: general description and implementation in MAMBA Image library” [en línia]. [Consulta: 7 abril 2020]. Disponible en: <http://www.cmm.mines-paristech.fr/~beucher/publi/HQ_algo_desc.pdf>
[6] “Proyectos Ágiles” [en línia]. [Consulta: 23 febrero 2020]. Disponible en: <https://proyectosagiles.org/desarrollo-iterativo-incremental/>
[7] “Search Job Salaries” [en línia]. [Consulta: 7 marzo 2020]. Disponible en: <https://www.payscale.com/research/ES/Job>
[8] Beucher, S.: Transformations résiduelles en morphologie numérique. Technical report of the Mathematical Morphology Center (2003).
[9] “MathWorks Help Center” [en línia]. [Consulta: 5 mayo 2020]. Disponible en: <https://es.mathworks.com/help/matlab/ref/tic.html>
[10] “MathWorks Help Center” [en línia]. [Consulta: 5 mayo 2020]. Disponible en: <https://es.mathworks.com/help/matlab/ref/toc.html>
[11] “The Berkeley Segmentation Dataset and Benchmark” [en línia]. [Consulta: 26 mayo 2020]. Disponible en: <https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/>