# Capítulo 3

# MÉTODO LIVE WIRE

Como ya se ha visto en el Capítulo 2, una de las formas de segmentación semiautomática de gran importancia en la actualidad es el ***método Live Wire*** o 'cable vivo', también conocido como **intelligent scissors** ('tijeras inteligentes'), el cual fue introducido por primera vez por *William. A. Barret* y *Eric. N. Mortensen* **[4]** en 1992, como solución intermedia para la realización de una segmentación eficiente que fusionara las ventajas del trazado manual de contornos y del cálculo computacional del proceso de segmentación automático **[2]**, **[3]**.

Aunque ya se profundizará más adelante en el total funcionamiento del programa (entradas que recibe, archivos de salida, etc.), se hace necesaria una breve explicación del método en el que se basa para ver cómo interviene el usuario en el proceso e introducir algunos conceptos que servirán para la completa inteligibilidad del mismo. Grosso modo se puede resumir el método Live Wire en los siguientes pasos:

**1)** El programa (realizado en **MATLAB**) recibe a la entrada, entre otros parámetros, la imagen que se va a segmentar.

**2)** A continuación se le pedirá al usuario que pinche con el ratón sobre un punto de la frontera del objeto que quiere segmentar. Este punto será el ***píxel semilla***, *s*.

**3)** Se procesará la imagen de entrada para obtener una matriz *L(p,q)*, denominado **mapa de coste local (MCL)**, que contiene la información del coste asociado de " ir desde un píxel *p* a su vecino *q* ".

**4)** A partir del mapa de coste local *L(p,q)* y del píxel semilla *s* se calculará, utilizando el *algoritmo de Dijkstra*, una nueva matriz *G(p)* o **mapa de coste acumulado (MCA)** que proporciona el camino mínimo entre el píxel semilla y cualquier otro píxel *p* de la imagen.

**5)** Seguidamente, hechos los cálculos anteriores, se le pide al usuario que pinche en otro punto de la imagen distinto al píxel semilla y perteneciente a la frontera del objeto a

segmentar, que determinará el píxel que será el otro extremo del camino a seguir por la frontera. Es decir, el programa obtiene el camino mínimo entre el píxel semilla inicial y el siguiente píxel pedido, denominado ***píxel posición*** *p**.

Cada uno de estos caminos que unen los píxeles *s* y *p** se denominan **segmentos Live Wire**. Este método hace que se disponga de una herramienta efectiva y a la carta para determinar el contorno de un objeto dentro de la imagen en función de la posición pinchada con el cursor del ratón.

**6)** El píxel posición se convierte en la nueva semilla (*s* <-- *p**).

**7)** Se repiten los pasos **4** a **6** hasta que el usuario pulse el segundo botón del ratón, momento en que la ejecución de programa habrá finalizado, bien porque el usuario obtuvo ya la frontera del objeto deseada, bien porque éste decidió terminar la segmentación en ese punto.

Es importante destacar que, en principio [1], en el proceso de segmentación se calcula un único **mapa de coste local** para toda la imagen, ya que no existe ninguna dependencia con ningún parámetro variable en el proceso, mientras que el **mapa de coste acumulado** hay que recalcularlo cada vez que cambie el píxel semilla.

## 3.1. MAPA DE COSTE LOCAL

Físicamente la matriz ***L(p,q)*** representa el coste local de pasar desde cualquiera de los 8-vecinos ***q*** del píxel ***p*** hasta él. Es decir, si un píxel del mapa de coste toma un valor X, significa que el coste local de sus 8-vecinos en llegar hasta él es X. El valor de *L(p,q)* en cada píxel se calcula como la suma ponderada de tres funciones **[4]**:

$$L(p,q) = w_Z\, f_Z\,(q) + w_G\, f_G\,(q) + w_D\, f_D\,(p,q) \qquad (3.1)$$

$f_Z$ : función detección de contorno

$f_G$ : función magnitud del gradiente

$f_D$ : función dirección del gradiente

$w_Z$, $w_G$, $w_D$ : pesos de cada una de las funciones de coste

A continuación se explicará cada una de las funciones anteriores. La ponderación de cada una de ellas se determinará posteriormente en un análisis experimental que se detalla en el Capítulo 5.

### 3.1.1. FUNCIÓN DETECCIÓN DE CONTORNO

Como ya se vio en el capítulo anterior, se utiliza el **método de Canny** por tratarse del detector de bordes más potente en la actualidad, lo que supone una modificación del método original en que se basa este proyecto. Para ello se hará uso de la función '*edge*' de MATLAB que toma la intensidad de una imagen I a la entrada, y **devuelve una imagen binaria BW, del mismo tamaño que I, con 1 en los píxeles donde se detectan bordes y 0 en el resto de ellos**.

---

[1] realmente no será así, ya que se han realizado modificaciones del método original

El formato de la función es el que se muestra a continuación, donde el método de segmentación será el de '*canny*':

BW = edge (I, 'segmentation method')

Los pasos principales del **método de Canny [5]** son:

- **Fase 1: SUAVIZADO**. Para eliminar ruido se convoluciona la imagen con un filtro gaussiano, óptimo en resolver el permanente conflicto entre reducir el ruido o detectar bien los bordes. Este filtro hace la imagen más homogénea y elimina los falsos contornos.

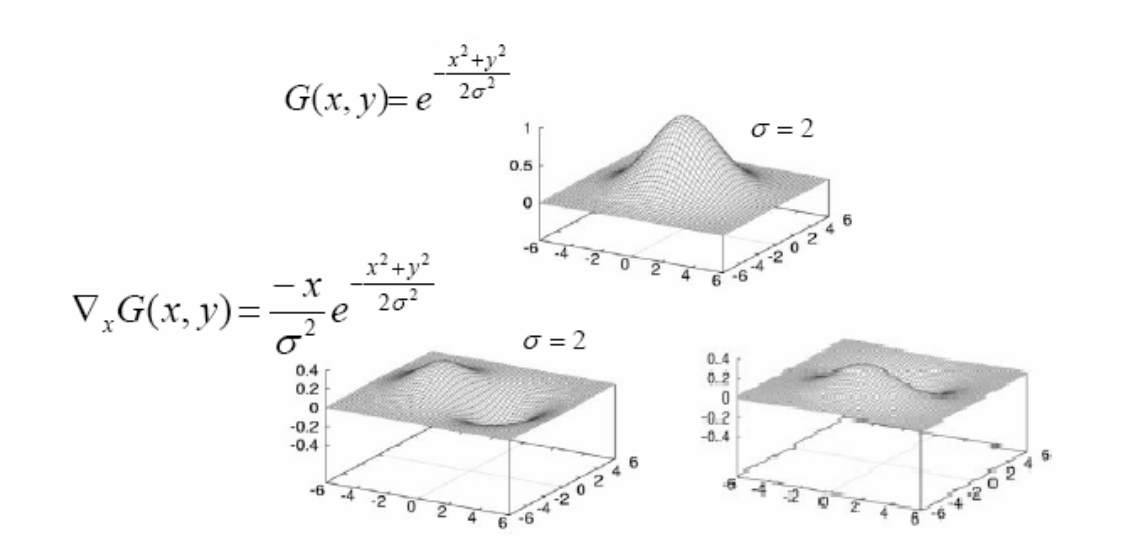



**Figura 3.1.** Representación de la distribución gaussiana bidimensional y de sus primeras derivadas (σ = 2).

Una posible implementación de un filtro gaussiano puede ser la siguiente:

$$\frac{1}{115}$$

| | | | | |
|---|---|---|---|---|
| 2 | 4 | 5 | 4 | 2 |
| 4 | 9 | 12 | 9 | 4 |
| 5 | 12 | 15 | 12 | 5 |
| 4 | 9 | 12 | 9 | 4 |
| 2 | 4 | 5 | 4 | 2 |

- **Fase 2: REALZADO**. A continuación se calcula el gradiente de la imagen suavizada usando una aproximación del gradiente de la función gaussiana, y se calcula su magnitud $|G| \approx |Gx| + |Gy|$ para determinar los píxeles donde se produce máxima variación.

- **Fase 3: DETECCIÓN**. Se calcula la dirección del gradiente $\theta_G = \arctan(Gy / Gx)$.

  **3.1. Supresión de no máximos.** Se seleccionan sólo los puntos que sean **máximos locales** en la dirección del gradiente y el resto se pone a 0.

  **3.2. Umbralización con histéresis.** Se usan dos umbrales, $T_1$ y $T_2$, para detectar los verdaderos bordes, de forma que para un píxel con valor $t$:

$$\begin{cases} Si \quad t < T_1 \quad el \ píxel \ NO \ es \ un \ borde \\ Si \quad T_1 \le t \le T_2 \quad el \ píxel \ es \ un \ borde \quad \Leftrightarrow \quad está \ conectado \ a \ un \ borde \\ Si \quad t > T_2 \quad el \ píxel \ es \ un \ borde \end{cases}$$

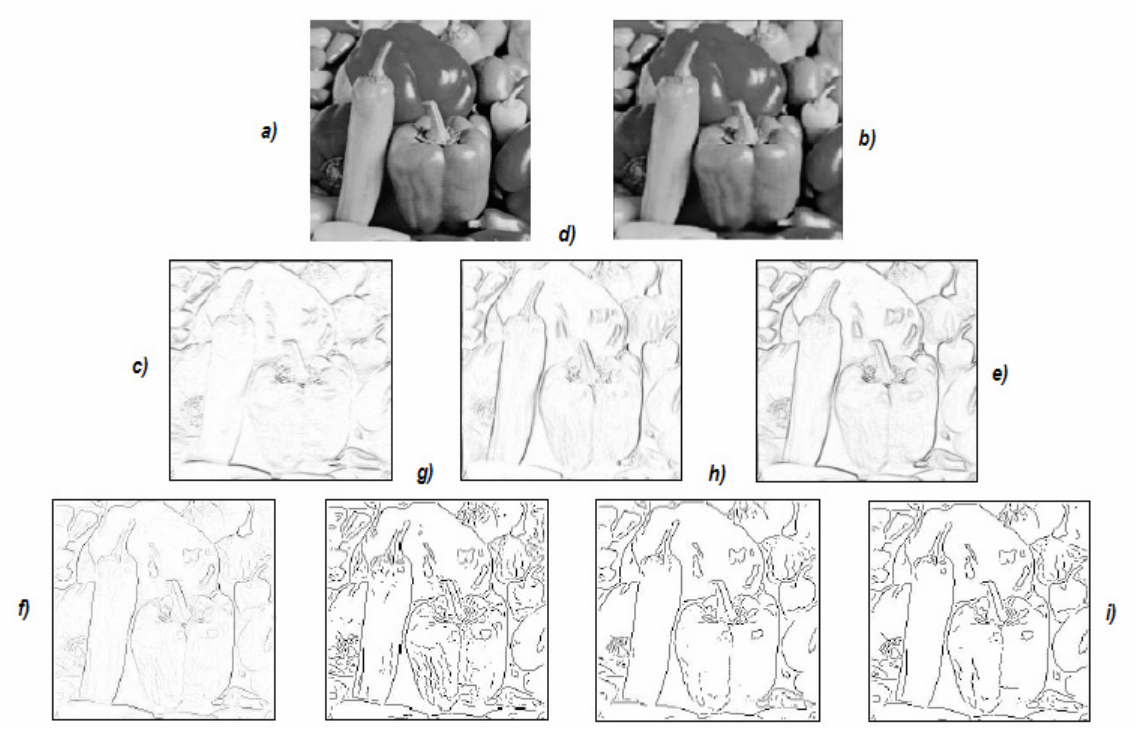



**Figura 3.2**. Aplicación del método de Canny. **a)** Imagen original. **Fase 1: b)** Suavizado. **Fase 2: c)** |Gy|. **d)** |Gx|. **e)** |G| = |Gx| + |Gy|. **Fase 3: f)** Eliminación de no máximos. **g)** Con umbral $T_1 = 30$. **h)** Con umbral $T_2 = 60$. **i)** Tras la doble umbralización, $T_1$ y $T_2$.

Como nuestra **imagen es a color**, se aplicará el *método de Canny* a los tres planos de color R, G y B. Cuando tengamos la función contorno de cada uno de ellos realizaremos una **OR** lógica para, de esta manera, poder detectar la existencia de un borde en uno cualquiera de los planos de color, lo cual implicaría la existencia de un borde en la imagen completa.

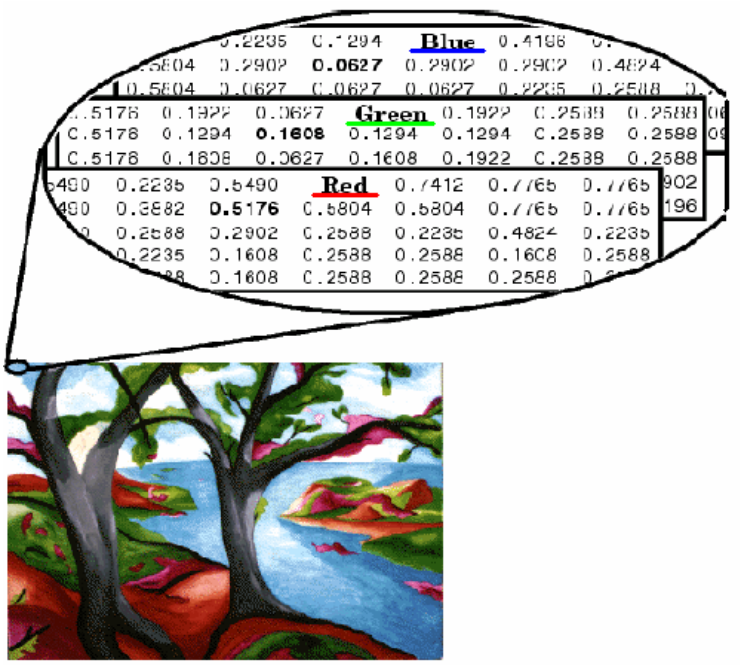



**Figura 3.3.** Representación de las tres componentes RGB en una imagen a color.

Hay que señalar que, como se detalló con anterioridad, la función '*edge*' devuelve 1 en los píxeles donde hay borde y 0 en los que no, mientras que nuestra intención es dar un coste nulo a lo píxeles de la frontera y alto (en este caso 1) a los que no lo son. Por consiguiente, se invertirá la imagen devuelta por '*edge*' para mantener la coherencia que se pretende en este trabajo.

### 3.1.2. FUNCIÓN MAGNITUD DEL GRADIENTE

La función de detección de contorno del apartado anterior crea una imagen binaria que determina si un píxel es un borde o no, pero no distingue el nivel o grado del mismo, es decir, no contempla la posibilidad de que un borde sea más fuerte o débil que otro. Para ello haremos uso de la **magnitud del gradiente [6]**.

Como ya se ha visto, el gradiente en imágenes digitales mide la diferencia de intensidad entre píxeles vecinos en alguna dirección, de forma que nos proporciona una correlación directa entre la "fuerza" del borde y un coste local. Si *Gx* y *Gy* representan los gradientes parciales de una imagen en las direcciones *x* e *y* respectivamente, entonces la magnitud del gradiente vendrá dada por

$$G = \sqrt{Gx^2 + Gy^2} \qquad (3.2)$$

En el programa se ha realizado la función '*grad_mag*' (Apéndice A) que llama a la función de MATLAB '*gradient*', la cual devuelve el gradiente numérico de la matriz que se le pasa a la entrada:

$$[\mathrm{Gx}, \mathrm{Gy}] = \mathrm{gradient}\ (\mathrm{matrix})$$

Una vez calculado el gradiente, se escala y se invierte de forma que **a mayor magnitud del mismo habrá un mayor cambio de intensidad y, por tanto, se asociará un coste bajo por tratarse de un borde**. Por el contrario, si el cambio de intensidad es bajo se asocia un coste alto pues nos encontramos en una zona homogénea de la imagen. Esto se puede traducir matemáticamente en la siguiente ecuación:

$$f_G = 1 - \frac{G}{\max(G)} \tag{3.3}$$

Como se advirtió en el apartado anterior, al tratar **imágenes a color** la operación hay que realizarla para los tres planos de color R, G y B, y la función de la magnitud del gradiente final será el mínimo valor entre los tres planos. De esta manera no se enmascara el hecho de que exista un borde muy significativo en un sólo plano de color y en los otros dos planos no se advierta **[w4]**. Esta decisión se justificará con imágenes en el Capítulo 5.

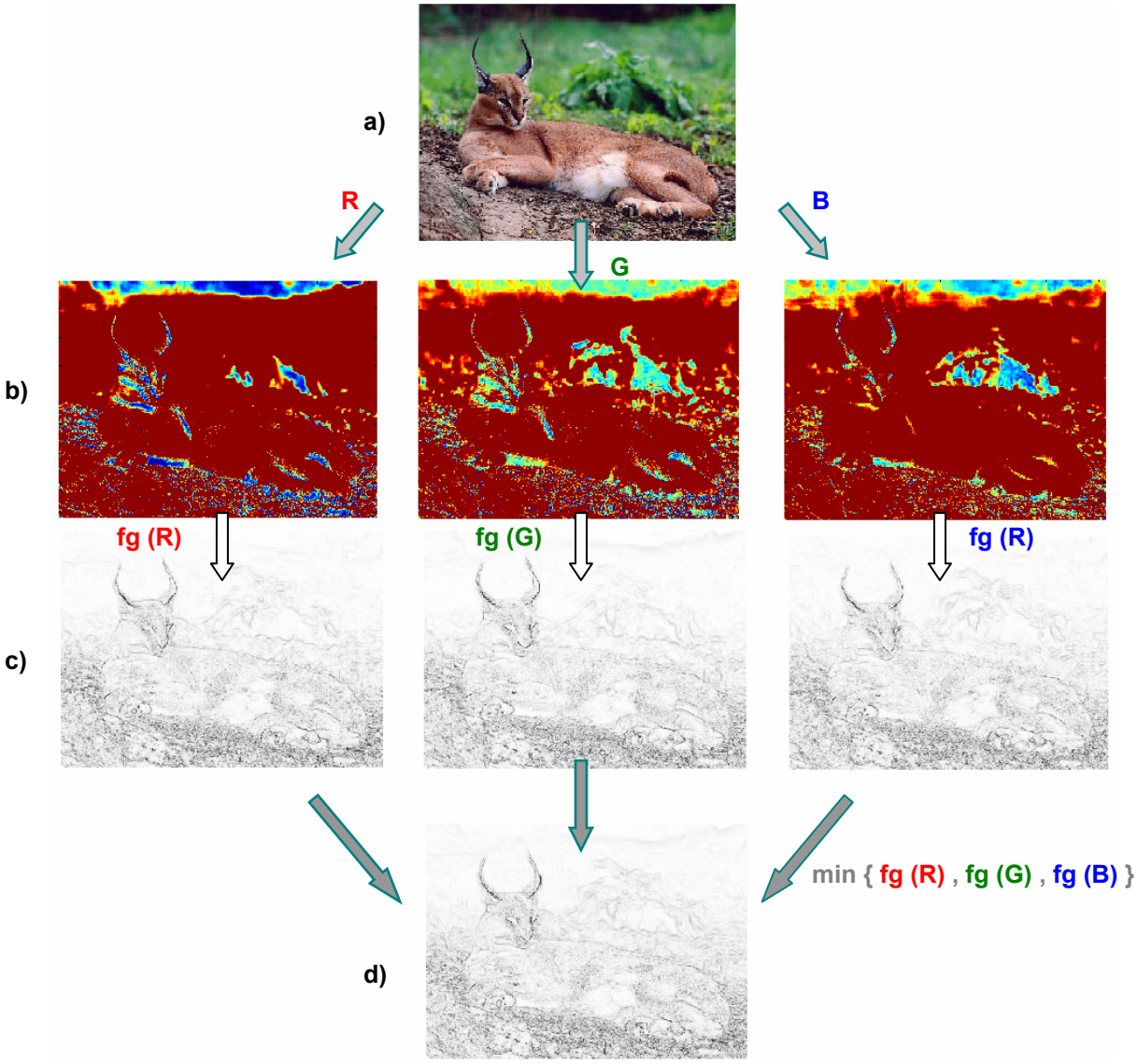



**Figura 3.4. a)** Imagen original a color. **b)** Componentes del plano RGB. **c)** Magnitud del gradiente de cada plano. **d)** Magnitud del gradiente de la imagen original.

### 3.1.3. FUNCIÓN DIRECCIÓN DEL GRADIENTE

La dirección del gradiente añade un término suave a la frontera asociando un alto coste para cambios rápidos de formas en la dirección de la misma. Esto añade más fiabilidad a la segmentación si los contornos se interrumpen o cambian su apariencia localmente. Por otra parte, un alto peso en este término no actúa muy bien en objetos con siluetas que oscilan rápidamente **[4]**.

La **dirección del gradiente [4]** es el vector unitario definido por las componentes horizontal y vertical del gradiente, *Gx* y *Gy*. Definiendo $\hat{D}$ ***(p)*** como el vector unitario normalizado perpendicular (rotado 90 grados en el sentido de las agujas del reloj) a la dirección del gradiente en el punto ***p***, esto es, $\hat{D}$ ***(p) = [ Gy(p), -Gx(p) ]***, la formulación del coste de la dirección del gradiente es

$$f_D = \frac{1}{\pi}\left\{ ar\cos\left[d_p(p,q)\right] + ar\cos\left[d_q(p,q)\right] \right\} \quad (3.4)$$

donde

$$\begin{aligned} d_p(p,q) &= \hat{D}(p)\cdot\hat{L}(p,q) \\ d_q(p,q) &= \hat{L}(p,q)\cdot\hat{D}(p) \end{aligned} \quad (3.5)$$

son los productos escalares de ***D*** y ***L*** normalizados y

$$\hat{L}(p,q) = \frac{1}{\|q-p\|}\begin{cases} q-p & si \quad \hat{D}(p)\cdot(q-p) \geq 0 \\ p-q & si \quad \hat{D}(p)\cdot(q-p) < 0 \end{cases} \quad (3.6)$$

es el vector enlace bidireccional normalizado entre los píxeles ***p*** y ***q***. Los enlaces entre píxeles pueden ser horizontal, vertical o diagonal, dependiendo de la posición del píxel ***p*** y la de su vecino ***q***, y apunta en la dirección que hace positivo el producto escalar entre $\hat{D}$ (**p**) y $\hat{L}$ (**p**,**q**), como se concluye de la **Ecuación 3.6**.

En la formulación anterior se observa que **se asocia un bajo coste si la dirección del enlace entre dos píxeles, *p* y *q*, es perpendicular, o casi perpendicular, al gradiente en *p*, ya que esto sería propio de la existencia de frontera**. Entonces el coste será alto cuando la dirección del gradiente de los dos píxeles es muy similar a la dirección del enlace entre ellos. En la **Figura 3.5** se puede observar con detalle el caso en que la función dirección del gradiente asocia en esa dirección un coste bajo, es decir, detecta un borde.

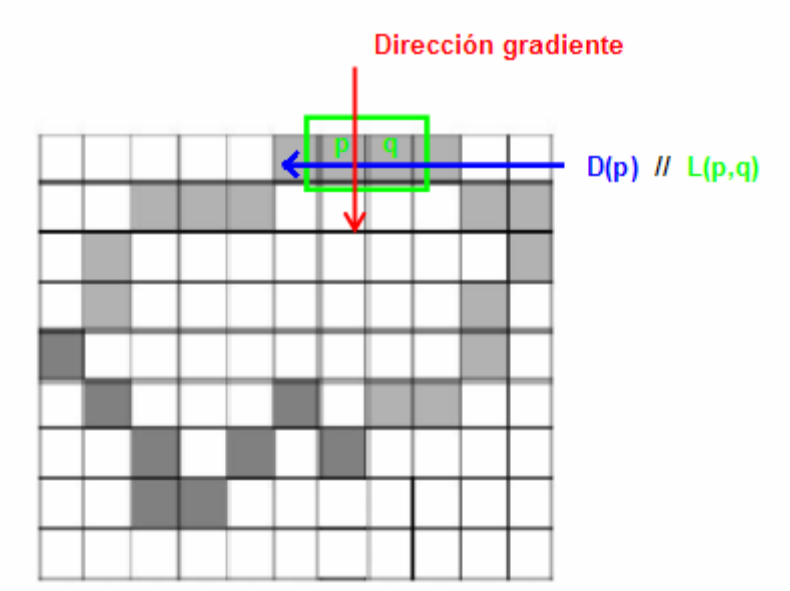



**Figura 3.5.** Direcciones de los vectores representativos en una imagen pixelada con contorno.

Se ha realizado una función en MATLAB denominada '*grad_dir*' (Apéndice A) que recibe la imagen a segmentar a la entrada y devuelve una matriz del mismo tamaño que la

anterior, pero donde cada celda es una matriz 3 x 3. Esto hace referencia a que ahora cada píxel tiene 8 costes para cada una de las 8 direcciones (8-vecinos de alrededor) que puede escoger cuando se quiere establecer un enlace entre él y un vecino.

La **Figura 3.6** representa la forma de la matriz que tenemos al calcular la dirección del gradiente. En ella se puede observar que cada píxel tiene ahora 8 costes, el elemento central se pone a infinito para indicar que nunca se tiene en cuenta (físicamente significaría el hecho de estar en un píxel y no "moverse" hacia ninguno de sus vecinos). Además los píxeles que forman parte del marco [2] de la imagen también tienen valor infinito hacia las direcciones prohibidas (direcciones que sobrepasan el tamaño de la imagen original).

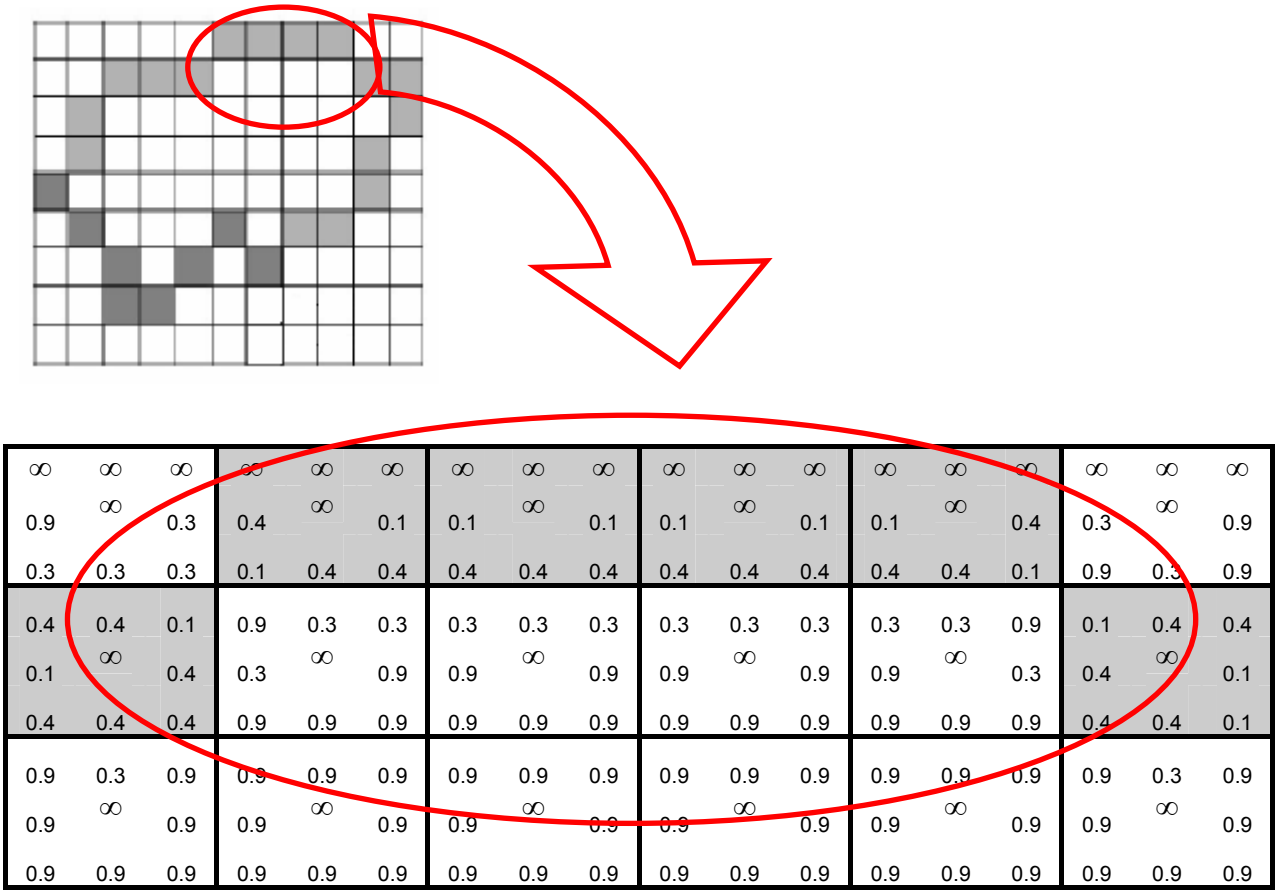


**Figura 3.6.** Forma matricial que adopta la dirección del gradiente y, en adelante, el mapa de coste local.

El cómputo de la dirección del gradiente **no cambia en imágenes a color**.

### 3.1.4. FORMACIÓN DEL MAPA DE COSTA LOCAL

Las tres funciones ponderadas vistas en los apartados anteriores conforman el mapa de coste local, de manera que es fácil advertir que éste tendrá la forma matricial mostrada previamente (**Figura 3.6**). El cometido de la formación del mapa de coste local es de la función '*mcl*', realizada en MATLAB y que se adjunta en el Apéndice A. A continuación se muestra una figura muy representativa de lo que hacen las tres funciones aplicadas por separado a una imagen, y su posterior unión.

[2] el marco de una imagen es el conjunto de píxeles que delimitan el tamaño de la misma

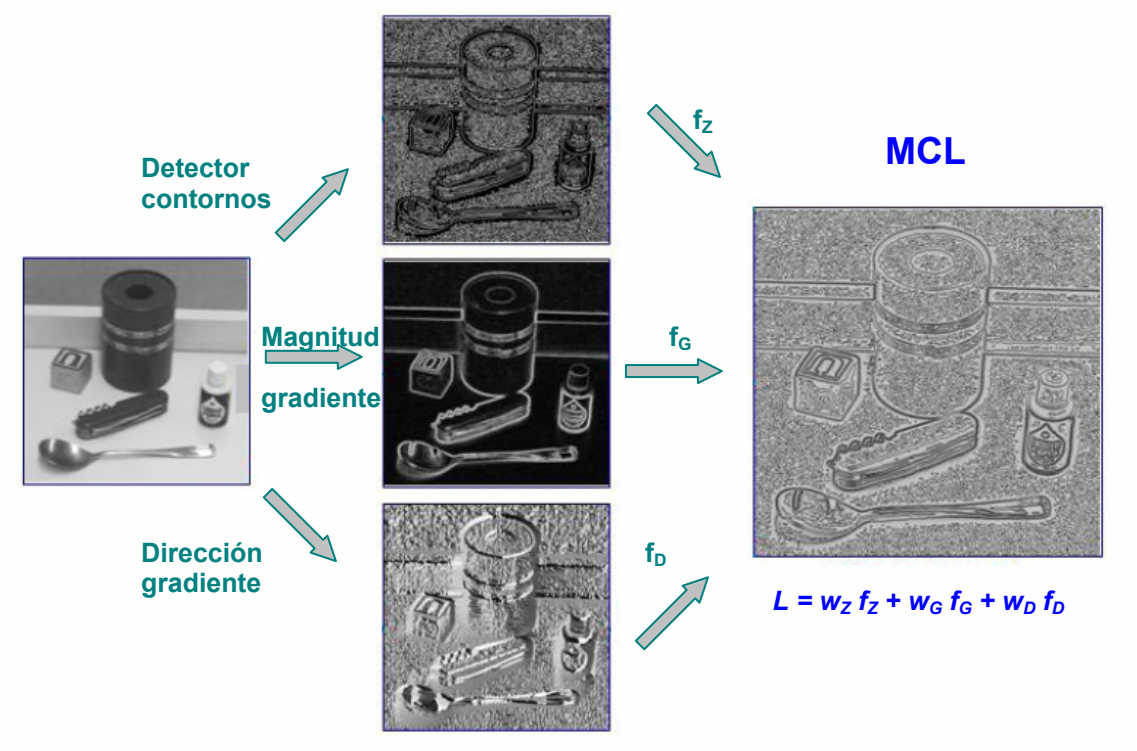



**Figura 3.7.** Formación del mapa de coste local.

## 3.2. MAPA DE COSTE ACUMULADO

El **mapa de coste acumulado** se obtiene aplicando el *algoritmo de Dijkstra* al mapa de coste local. Esto hace que la imagen se defina como un grafo orientado caracterizado por su función de coste y el problema de la segmentación queda reducido a la búsqueda del camino óptimo entre dos nodos del grafo.

### 3.2.1. ALGORITMO DE DIJKSTRA

El algoritmo de Dijkstra es una herramienta muy eficaz para encontrar un camino mínimo desde un nodo al resto de nodos que conforman el grafo. Se parte de un grafo formado por nodos y arcos dirigidos, cada uno con un peso, donde se diferenciará un nodo origen del resto. Inicialmente se considera que hay un subconjunto P que contiene únicamente a este nodo inicial.

El algoritmo es el siguiente **[w5]**:

**1)** Se eligen todos los nodos alcanzables desde el subconjunto P.
**2)** De todos esos nodos se escoge aquel que tenga un peso menor en su arco, y pasaría a formar parte del subconjunto P.
**3)** Se repiten **1** y **2** hasta que todos los nodos del grafo pertenezcan a P, es decir, se haya encontrado un camino mínimo desde el nodo inicial al resto.

Las siguientes Figuras, obtenidas a través de un *applet de Java* **[w6]**, muestran un ejemplo que implementa el algoritmo comentado. Se marcará en **azul** el **nodo inicial**, mientras que los **nodos** y **arcos candidatos** se encuentran en **rojo** y los **nodos elegidos** como pertenecientes al subconjunto P en **amarillo**.

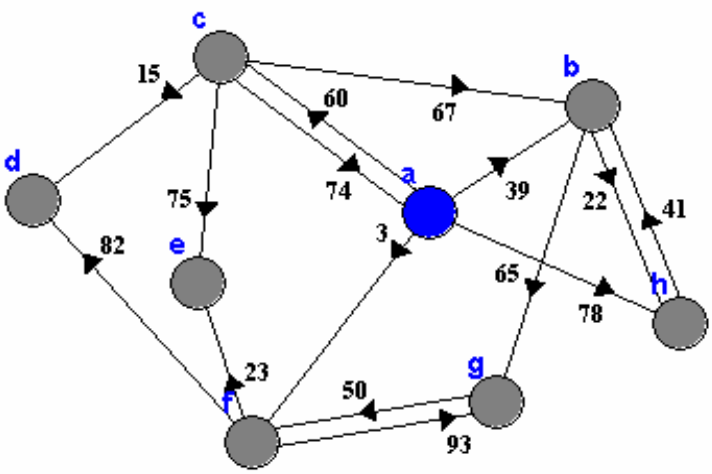



**Figura 3.8.** Representación del grafo al que se le aplicará el Algoritmo de Dijkstra (I).

Los nodos **b**, **c**, **f** y **h** son los únicos que se conectan con el nodo inicial **a**, de forma que son los candidatos (figura izquierda). De ellos se elige el **f** ya que es el de menor peso (figura derecha).

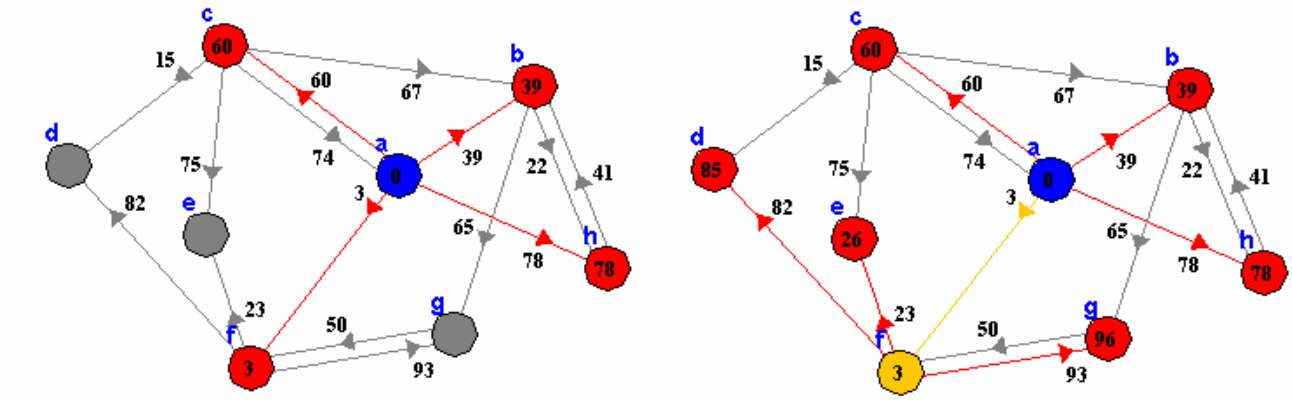



**Figura 3.9.** Algoritmo de Dijkstra (II).

Ahora el subconjunto P está formado por **a** y **f**, y el **resto** de nodos del grafo están conectados a él. Se escogerá el **e** ya que es el que acumula menor peso hasta llegar al inicial (izquierda). Continuando con el procedimiento el siguiente en unirse es el nodo **b** (derecha).

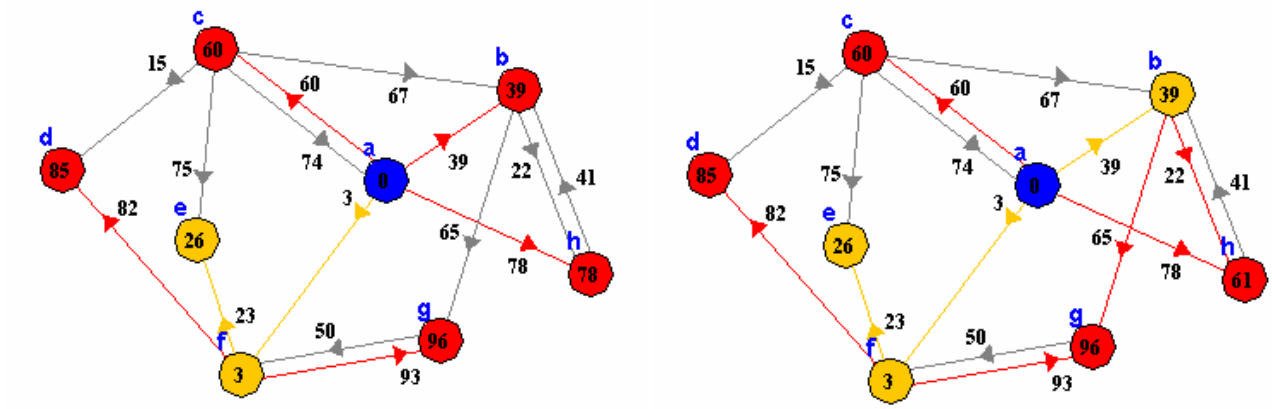



**Figura 3.10.** Algoritmo de Dijkstra (III).

Siguiendo con el procedimiento descrito se van añadiendo el resto de nodos al subconjunto P, siempre escogiendo el que tenga un menor peso en el camino hacia el nodo inicial (**Figuras 3.11** y **3.12**).

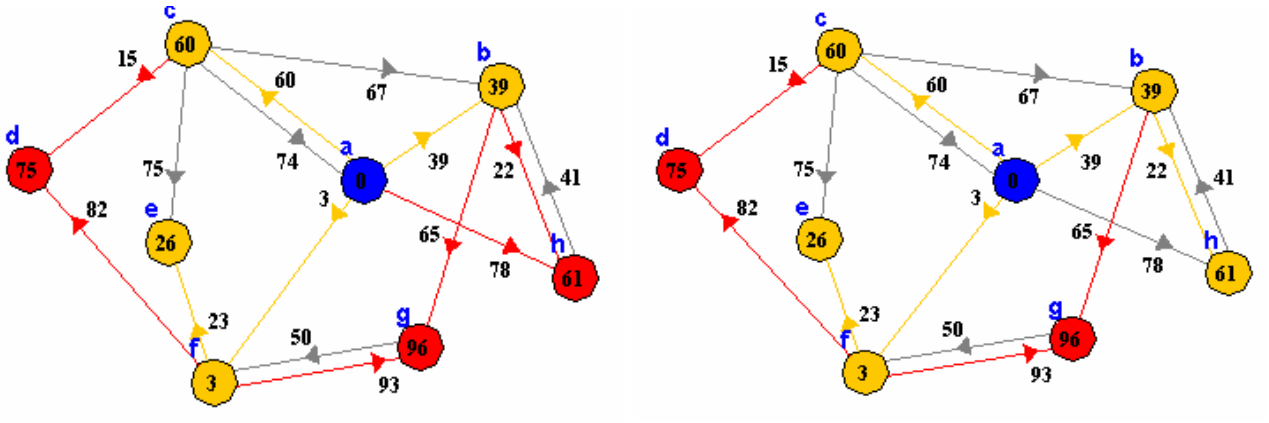



**Figura 3.11.** Algoritmo de Dijkstra (IV).

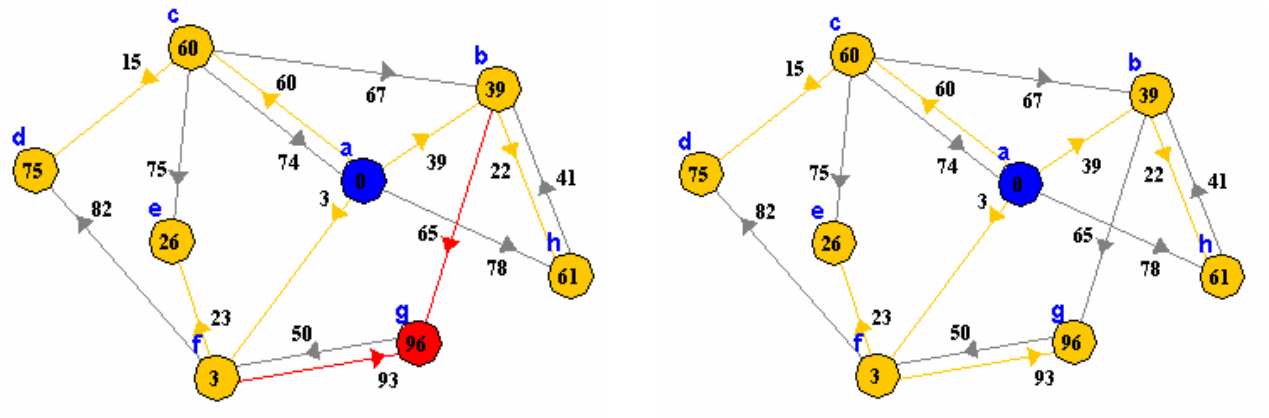



**Figura 3.12.** Algoritmo de Dijkstra (V).

Se observa como todos los nodos del grafo quedan conectados, de alguna manera, con el nodo inicial, existiendo así siempre un camino mínimo desde este nodo inicial hacia todos ellos.

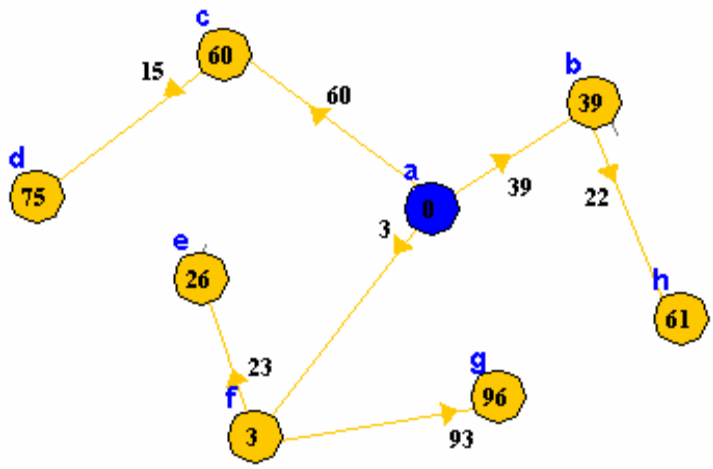



**Figura 3.13.** Algoritmo de Dijkstra (VI).

### 3.2.2. APLICACIÓN DE DIJKSTRA EN LA SEGMENTACIÓN DE IMÁGENES

*Falcao* y *Udupa* con su Live Wire y *Mortensen* y *Barrett* con su Intelligent Scissors fueron los primeros en introducir interactivamente el *algoritmo de Dijkstra* para el proceso de segmentación de imágenes **[7]**.

La función, realizada en MATLAB, encargada de calcular el mapa de coste acumulado es '*livewire*' (Apéndice A). Recibe a la entrada el mapa de coste local, que sería el grafo con los 8 pesos por cada píxel, y el píxel semilla pinchado por el usuario, que análogamente será el nodo inicial. Para entender la analogía total hace falta introducir algunas matrices que se utilizan en la función:

***expanded***: matriz del mismo tamaño del mapa de coste local que tiene valor 1 en los píxeles ya expandidos (que pertenecen al subconjunto P comentado en el apartado anterior y, por tanto, ya han alcanzado su valor definitivo) y 0 en el resto.

***active***: matriz del mismo tamaño del mapa de coste local que tiene valor 1 en los píxeles candidatos a expandirse (candidatos a pertenecer al subconjunto) y 0 en el resto.

***flechas***: matriz bidimensional que da como salida '*livewire*', y que informa sobre el píxel que sigue al actual para ir formando el camino mínimo deseado.

A continuación se muestra un ejemplo para explicar cómo se ha aplicado el *algoritmo de Dijkstra*. Se tiene una imagen de un determinado tamaño, aunque por simplicidad sólo se considerará un mapa de coste local 3 x 3, y que el píxel semilla será el elemento central (esto, obviamente, no es siempre así). Los valores dados en los arcos de los píxeles tampoco se corresponden con valores reales, de hecho los valores de un mapa de coste local pertenecen al rango 0 a 1, de manera que el uso de enteros es para dar mayor agilidad visual en la explicación. De esta forma, al mapa de coste local de la imagen viene dado por los costes que cada arco de un píxel concreto tiene sobre sus vecinos, y el mapa de coste acumulado estará formado por los valores que hay dentro de ese píxel, cuyo significado es el coste de ir desde él hacia la semilla.

En la **Figura 3.14** (derecha) se esquematiza en un grafo de píxeles una típica matriz correspondiente a un mapa de coste local. El píxel **semilla** está marcado en **celeste**. Inicialmente el mapa de coste acumulado tiene todos sus elementos a infinito, mientras que las matrices auxiliares, ***expanded***, ***active*** y ***flechas***, son matrices nulas (todos sus elementos a 0).

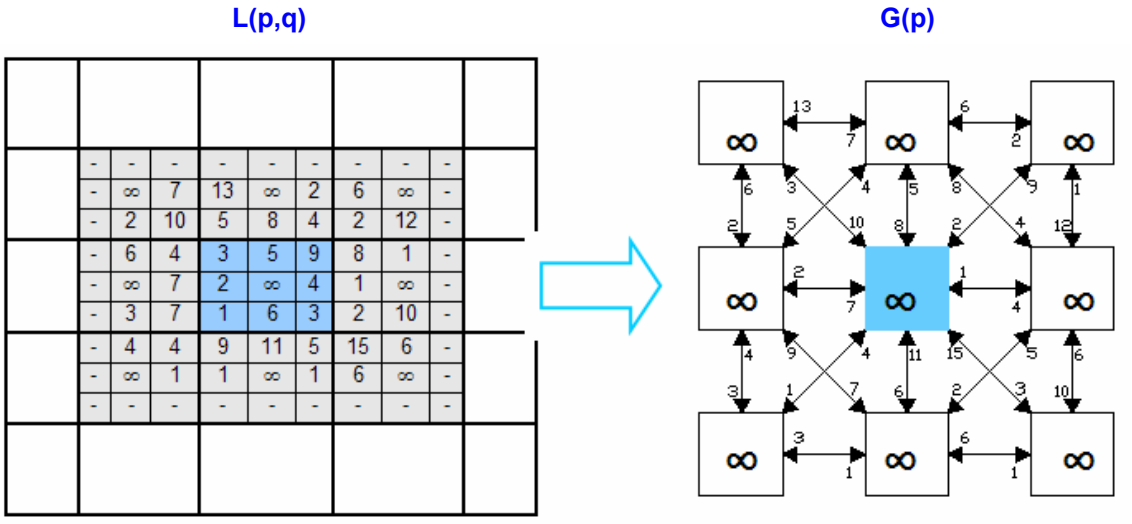
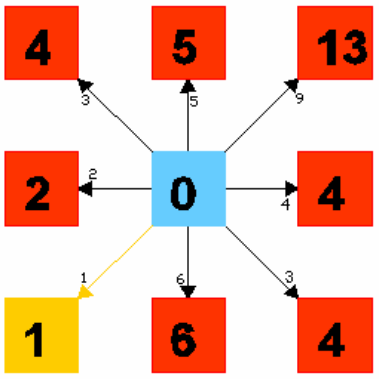



**Figura 3.14.** Paso del mapa de coste local matricial al mapa de coste local gráfico. L(p,q) y G(p) son las matrices de costes, local y acumulado, con las que trabaja 'livewire'.

El proceso paso a paso de la función '*livewire*' es el siguiente:

**(I)** En primer lugar se marca la semilla como expandida, mientras que todos sus vecinos (**candidatos** a expandirse) se hacen pertenecer a la lista activa a la vez que los actualizamos. Para ello se les asigna como valor su propio coste local escalado por la distancia Euclídea [3]. El siguiente píxel a expandir será el marcado como **amarillo** en la figura (píxel de mínimo coste de los que pertenecen a la lista activa).




***expanded***

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

***active***

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

***flechas (índices fila)***

$$\begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

***flechas (índices columna)***

$$\begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

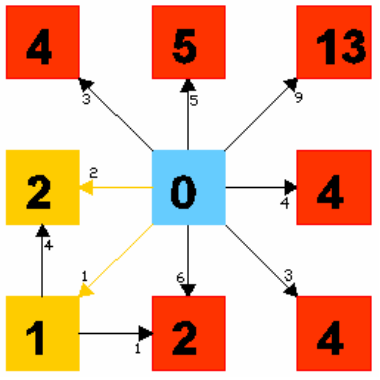
**Figura 3.15.** Obtención del mapa de coste acumulado, G(p) (I).

La expansión consiste en actualizar los vecinos que pertenezcan a la lista activa del píxel en cuestión. El valor de estos píxeles cambiará sólo si la suma del coste acumulado del píxel que se está expandiendo más el coste local propio de cada píxel es menor que el coste acumulado de cada uno. Esto no es mas que la matriz del mapa de coste acumulado *G(p)*, se forma según:

$$G(q) \quad = \quad \min\left\{ G(q)_{antiguo} \, , L(p,q)_{escalado} + G(p) \right\} \tag{3.7}$$

**(II)** En la **Figura 3.16** se observa como al actualizar los valores del mapa de coste acumulado, el píxel que está debajo de la semilla cambia su valor, ya que "cuesta menos llegar a él" a través del que tiene valor 1. Entonces se actualizan también las matrices flechas.

El elemento (i,j) de la matriz ***flecha (índices de fila)*** proporciona el índice de la fila de la posición siguiente a dicho elemento. Análogamente ***flecha (índices de columna)*** determinará el índice de la columna del píxel siguiente.



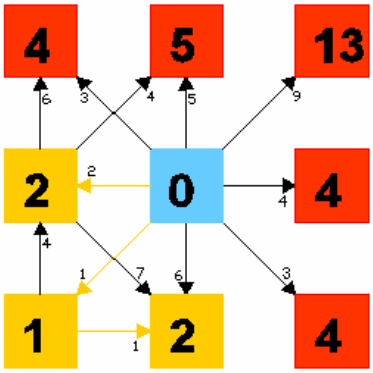

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 3 & 2 \end{bmatrix} \qquad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 1 & 2 \end{bmatrix}$$

**Figura 3.16.** Obtención del mapa de coste acumulado, G(p) (II).

[3] los píxeles que comparten un sólo vértice con el píxel a expandir (los de la diagonal) multiplican su coste local por $\sqrt{2}$

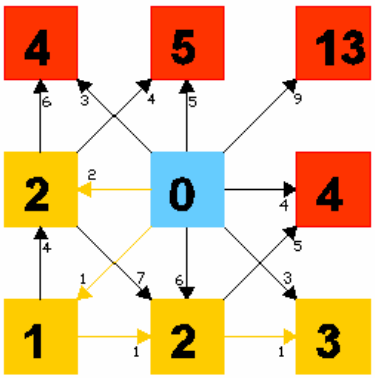
**(III)** Cuando dos o más píxeles de la lista activa tienen el mismo coste acumulado da igual cuál se escoja. En este caso se ha marcado el píxel a la derecha de la semilla como expandido y se actualizan los valores del mapa de coste acumulado, que en este paso no sufren cambios. Se observa que el próximo en expandirse será el que está bajo la semilla.




$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 3 & 2 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 1 & 2 \end{bmatrix}$$

**Figura 3.17.** Obtención del mapa de coste acumulado, G(p) (III).

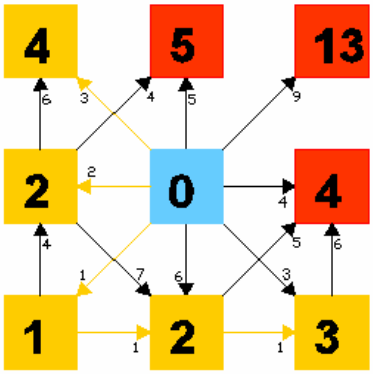
**(IV)** A continuación se marca el píxel bajo la semilla como expandido y se retira de la lista activa, esto significa que tendremos determinado cómo ir desde él hasta la semilla por medio de la matriz bidimensional ***flechas***. Luego se actualizarán los valores del mapa de coste acumulado (en este caso cambia el valor del píxel de la esquina inferior derecha).




$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 3 & 3 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 1 & 2 \end{bmatrix}$$

**Figura 3.18.** Obtención del mapa de coste acumulado, G(p) (IV).

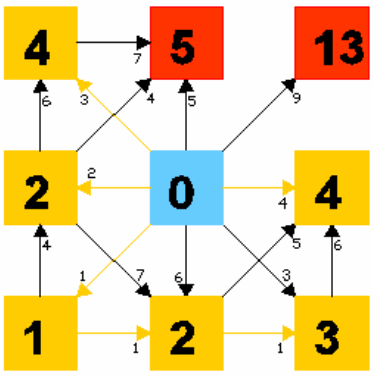
**(V)** Se marca el píxel que se determinó anteriormente como el mínimo de la lista activa. Se procede con la actualización de costes y se observa que el mapa de coste acumulado no sufre cambios.




$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 3 & 3 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 1 & 2 \end{bmatrix}$$

**Figura 3.19.** Obtención del mapa de coste acumulado, G(p) (V).

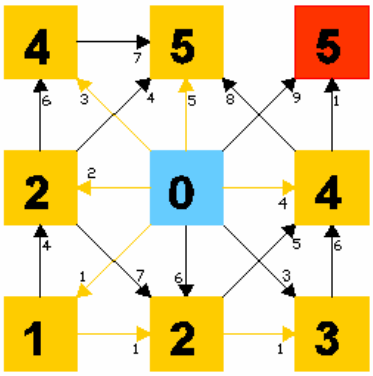
En las siguientes figuras se sigue mostrando el proceso de la formación del mapa de coste acumulado y las modificaciones de las matrices que nos ayudan a que éste funcione correctamente.




$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 3 & 3 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 1 & 2 \end{bmatrix}$$

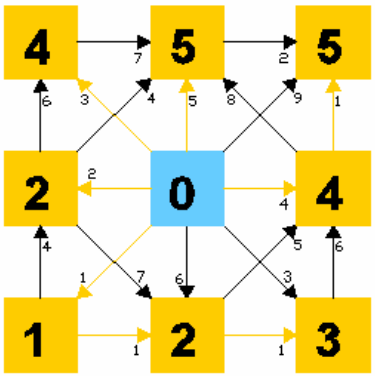
**Figura 3.20.** Obtención del mapa de coste acumulado, G(p) (VI).

**(VII)** En este paso el valor del píxel de la esquina superior derecha sufre una modificación, ya que "cuesta menos" llegar a él a partir del píxel a la derecha de la semilla.




$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 3 & 3 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 3 \\ 2 & 0 & 2 \\ 2 & 1 & 2 \end{bmatrix}$$

**Figura 3.21.** Obtención del mapa de coste acumulado, G(p) (VII).



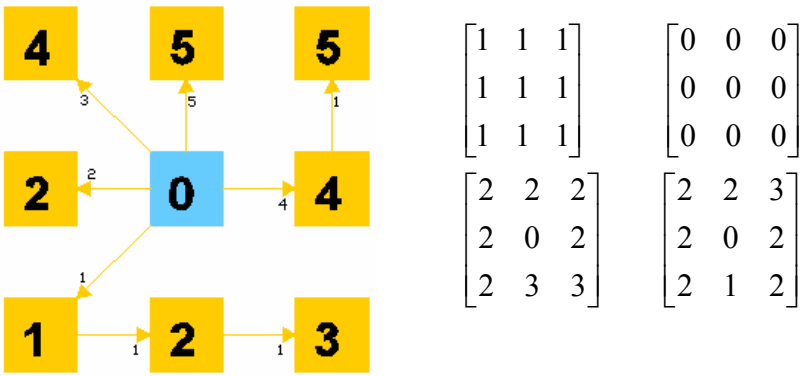

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 3 & 3 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 3 \\ 2 & 0 & 2 \\ 2 & 1 & 2 \end{bmatrix}$$

**Figura 3.22.** Obtención del mapa de coste acumulado, G(p) (VIII).

El proceso termina cuando todos los píxeles de la imagen quedan expandidos, esto es que la matriz ***expanded*** tiene todos sus elementos a 1, o lo que es lo mismo, no exista ningún píxel en la lista activa candidato a expandirse, es decir, la matriz ***active*** tenga todos sus elementos a 0.

Se ha observado durante todo el proceso anterior la simplificación del contenido de las figuras explicativas. Por ejemplo, se han suprimido los arcos de llegada a los píxeles ya expandidos, pues una vez expandidos ya poseen su camino mínimo hacia el píxel semilla, de manera que éstos no van a afectar a lo que quede del proceso.

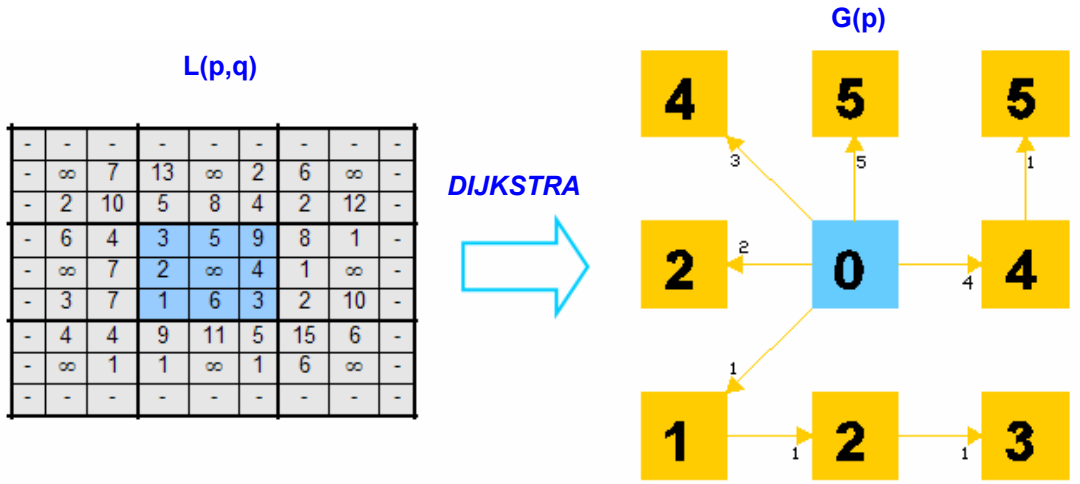
Finalmente, advertir cómo se ha creado la matriz ***flechas*** que será la salida de la función '*livewire*'. En la **Figura 3.23** se observa como ésta define el camino a seguir desde el píxel escogido como semilla hasta cualquiera de los píxeles restantes. Por tanto, este procedimiento **resuelve el problema de encontrar el camino mínimo desde un píxel al resto de píxeles de la imagen que determinará la segmentación de la misma**.




$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 2 & 2 \\ 2 & 0 & 2 \\ 2 & 3 & 3 \end{bmatrix} \quad \begin{bmatrix} 2 & 2 & 3 \\ 2 & 0 & 2 \\ 2 & 1 & 2 \end{bmatrix}$$

**Figura 3.23.** Obtención del mapa de coste acumulado, G(p) (IX).




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - | - |
| - | ∞ | 7 | 13 | ∞ | 2 | 6 | ∞ | - |
| - | 2 | 10 | 5 | 8 | 4 | 2 | 12 | - |
| - | 6 | 4 | 3 | 5 | 9 | 8 | 1 | - |
| - | ∞ | 7 | 2 | ∞ | 4 | 1 | ∞ | - |
| - | 3 | 7 | 1 | 6 | 3 | 2 | 10 | - |
| - | 4 | 4 | 9 | 11 | 5 | 15 | 6 | - |
| - | ∞ | 1 | 1 | ∞ | 1 | 6 | ∞ | - |
| - | - | - | - | - | - | - | - | - |

**Figura 3.24.** Algoritmo de Dijkstra como herramienta para obtener la matriz del mapa de coste acumulado a partir de la matriz del mapa de coste local.

## 3.3. PROBLEMAS ENCONTRADOS Y MODIFICACIONES

En este apartado se comentan los problemas que se han ido produciendo a medida que el programa se desarrollaba y no se obtenían las prestaciones esperadas. Igualmente se explican las soluciones aportadas para cada uno de estos problemas. Aunque estas soluciones no son más que modificaciones adicionales del método, es importante señalar que estos problemas no son particulares de este algoritmo, sino que pueden aparecer en cualquier tipo de método de segmentación y, por tanto, son igualmente planteables en revisiones futuras del método o en líneas de investigación similares a la aquí descrita.

Aunque se señalan dos problemas, realmente hay uno principal: el tiempo de segmentación. Mientras que el segundo problema es derivado de éste, es decir, de alguna manera la solución adoptada para el primer problema estaba incompleta e hizo falta una revisión de la misma. Aún así, por mayor claridad, se explicarán los dos por separado.

### 3.3.1. PROBLEMA 1: EL TIEMPO DE SEGMENTACIÓN

La principal dificultad encontrada en la implementación en MATLAB del método Live Wire es que **el tiempo que tarda el programa realizado en segmentar el objeto de la imagen es excesivo**. Se obtenían tiempos de varios minutos que son inaceptables tanto por la carga computacional que supone en el ordenador como por el tiempo de espera que le supone al usuario. Las razones de este excesivo tiempo son dos:

**1)** La función que calcula la **dirección del gradiente** de una imagen tarda un tiempo considerable en realizarse. Como ya se vio, por cada píxel de la imagen se deben hallar hasta ocho costes, cuyo cálculo además no es inmediato si no que requiere de una formulación bastante engorrosa que, en definitiva, carga demasiado al programa.

**2)** Por otra parte, la **aplicación del algoritmo de Dijkstra** para encontrar el camino más corto de un píxel a otro, enlentece también bastante el programa si el tamaño de la imagen a segmentar es elevado.

Es claro concluir que a mayor tamaño de una imagen mayor tiempo de segmentación. De hecho, se comprobó experimentalmente que el programa tardaba aproximadamente ***4 ms/píxel*** (ver **Tabla 3.1**) en realizar la segmentación. Esto supone para imágenes de 300 x 225 (tamaño típico de las imágenes médicas con las que se ha trabajado) un tiempo entre 4 y 6 minutos, donde el 85 % del tiempo se corresponde al tiempo computacional del mapa de coste local, es decir, a calcular el mapa de coste local de la imagen completa. El 15 % restante se divide entre el tiempo de meditación del usuario y el tiempo de cálculo del mapa de coste acumulado.

La consigna para resolver esta limitación parece bastante clara: **operar con "algo" de tamaño más reducido.**

**SOLUCIÓN. - CREAR SUBIMAGEN**

Como la segmentación final está compuesta por varios segmentos, cada uno formado a partir del par de píxeles introducidos por el usuario (píxeles *semilla* y *posición*), se puede considerar que el rectángulo que determinan estos dos puntos contiene toda la información necesaria para la segmentación (a priori). Así que para no cargar el programa se ideó la creación de una subimagen con cada par de píxeles pinchados (tal como se deduce de la **Figura 3.25**).

Para ello la función '*mcl*', encargada de hallar el mapa de coste local, llama a la función '*subimagen*' (Apéndice A) que crea y devuelve la subimagen definida por los dos píxeles que se le pasan a la entrada. Con esto se reduce gratamente el tiempo de segmentación, ya que sólo a esta subimagen será a la que se le aplique todo el proceso de cálculo de mapas de coste local y acumulado, y la representación del camino más corto.

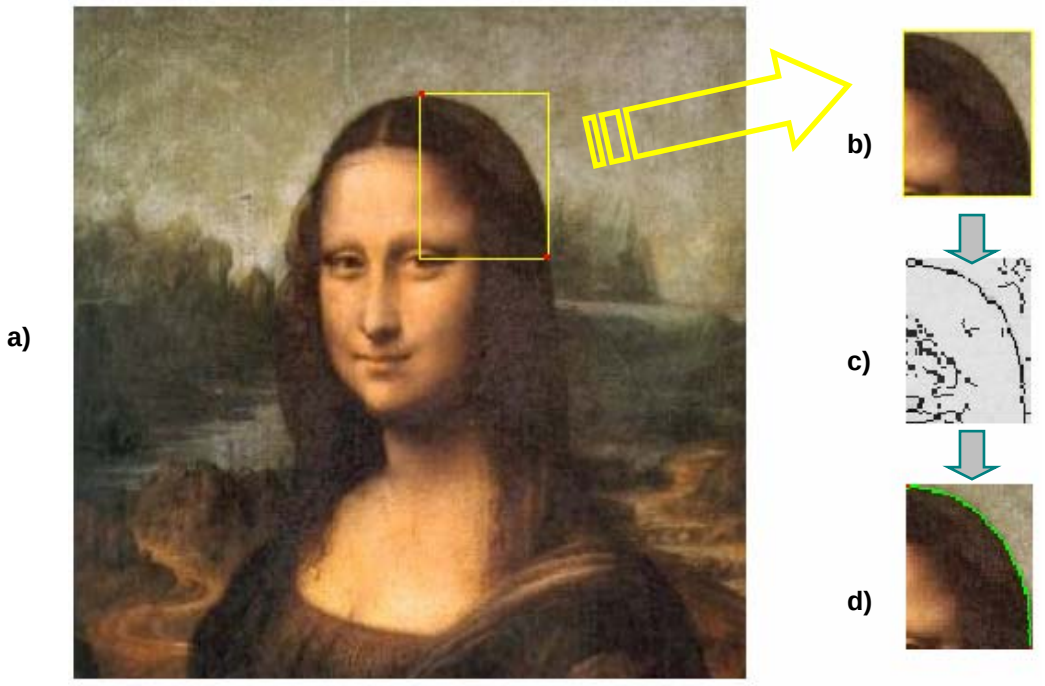



**Figura 3.25.** Proceso de creación de subimagen y aplicación a ésta del método. **a)** Figura original. **b)** Subimagen creada. **c)** Mapa de cote local de la subimagen. **d)** Segmento Live Wire correspondiente.

Se observa entonces como la segmentación de la imagen se va produciendo en varios segmentos que recorren el contorno que ha quedado encuadrado en la subimagen (**Figura 3.26**). Esta solución implica un tiempo de computación mucho menor (ver **Tabla 3.1**) que realizar el mapa de coste local a la imagen completa e iniciar luego el proceso de segmentación. **Con esta solución se crean tantos mapas de coste local y de coste acumulado como segmentos Live Wire fuesen necesarios para la segmentación completa de la imagen (modificación del método original).**

| imagen | nº puntos | tamaño (px) | t. segmentación sin subimágenes | | t. segmentación con subimágenes (seg) |
|---|---|---|---|---|---|
| | | | $t_{MCL}$ (seg) | resto (seg) | |
| **ampolla.jpg** | 4 | 306 x 226 | 280 | 30 | 50 |
| **image5.jpg** | 7 | 300 x 225 | 275 | 68 | 150 |
| **image14.jpg** | 7 | 300 x 219 | 286 | 87 | 180 |
| **image22.jpg** | 7 | 300 x 219 | 270 | 90 | 165 |
| **image23.jpg** | 7 | 300 x 219 | 267 | 90 | 180 |
| **serval.jpg** | 12 | 612 x 448 | 1180 | 350 | 650 |
| **winterice.jpg** | 8 | 800 x 600 | 2130 | 225 | 340 |
| **manhattan.jpg** | 8 | 1024 x 768 | - | - | 250 |

(4)

**Tabla 3.1.** Tiempos de segmentación con utilización de subimágenes (margen = 20) y sin ella.

[4] ver imágenes en Apéndice B

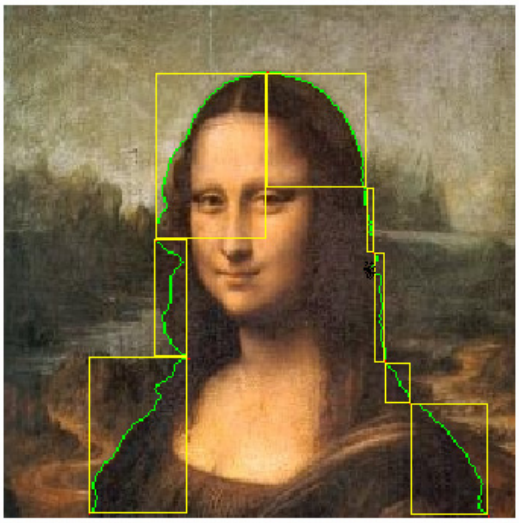


**Figura 3.26.** Representación de las distintas subimágenes creadas durante la segmentación.

### 3.3.2. PROBLEMA 2: CONTORNOS NO ENCUADRADOS

La solución propuesta para el problema anterior optimiza el proceso en tiempo pero no en calidad. Es fácil advertir que todo lo que no quede dentro de la subimagen creada no es computado, es decir, **si parte de la frontera que queremos segmentar no se encuadra en la subimagen creada la segmentación no es correcta** (el proceso tiende a buscar un borde aunque éste no sea el deseado por el usuario). La figura siguiente demuestra este hecho.

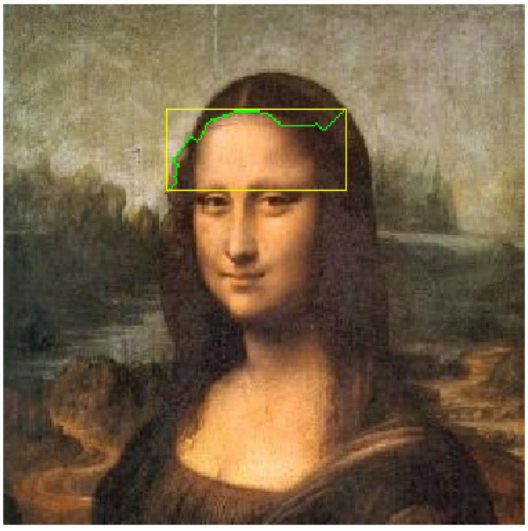


**Figura 3.27.** Problema de detección de contorno si éste no pertenece a la subimagen.

**SOLUCIÓN. - APLICAR UN MARGEN DINÁMICO**

Es sencillo pensar que si aumentamos el tamaño de la subimagen conseguiremos que se detecte un mayor porcentaje de los contornos no encuadrados, el problema es saber cuánto aumentar la subimagen (valor del parámetro ***margen***).

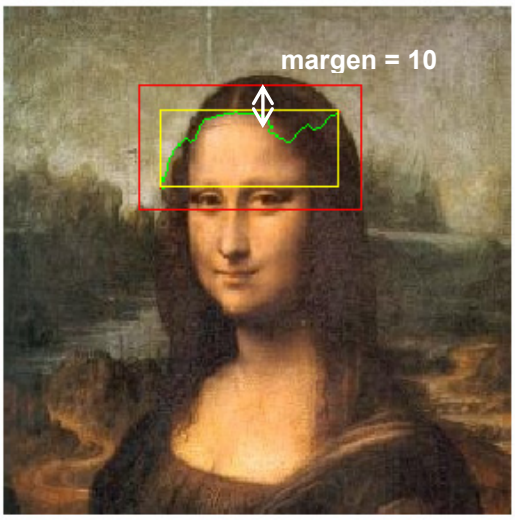
Se pensó que el ***margen*** podría ser una cantidad fija, pero esto limitaba demasiado el radio de acción del usuario. Luego se dedujo que si fuese igual a las distancias vertical y horizontal de la subimagen era muy difícil que un contorno quedara fuera del encuadre, pero

esto hacía que el tiempo de segmentación aumentara considerablemente (se está doblando el tamaño de la subimagen). Además en la mayoría de los casos coger el doble de una subimagen para segmentar es una sobreprevisión.

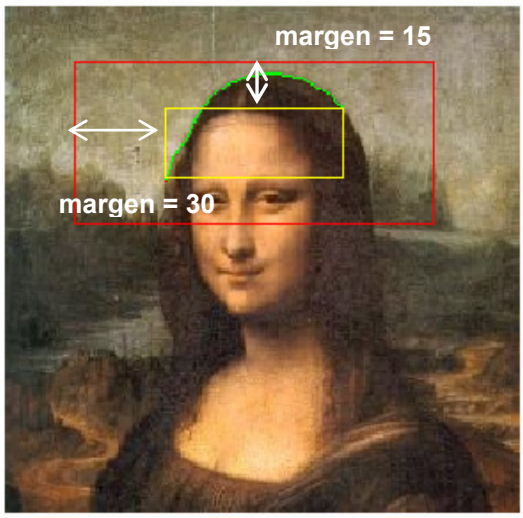
Como solución de compromiso se adoptó que el valor del margen fuese elegido por el usuario, por lo que se convirtió en un parámetro. Así que la idea es **escoger un valor adecuado para obtener la segmentación deseada en el menor tiempo posible**. Si se sabe que el objeto a segmentar tiene "contornos muy picudos" es lógico poner un valor alto en el margen aún conociendo que nos llevará más tiempo del esperado realizar la segmentación. Si por el contrario sabemos que el objeto tiene contornos de poco radio de curvatura se asignará un margen pequeño, incluso nulo.

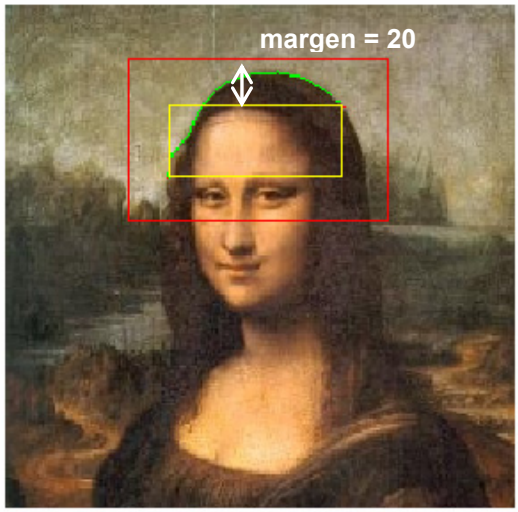
Es importante señalar que influye mucho la experiencia que tenga el usuario, es decir, a veces puede no ser necesario poner un valor muy alto de margen si se pincha en lugares estratégicos del objeto.

a)




b)




c)

**Figura 3.28.** Distintas soluciones comentadas (diferentes márgenes). **a)** Margen fijo igual a 10: segmentación incorrecta. **b)** Márgenes variables igual a 15 y 30 (dimensiones dobladas): segmentación correcta pero excesivo tiempo (65 seg). **c)** Margen como parámetro: correcta segmentación y tiempo mejorado (38 seg).

## 3.4. RESUMEN

- La **función detección del contorno** devuelve una imagen binaria con un 1 en los píxeles donde se detectan bordes y 0 en los que no.

- La **función magnitud del gradiente** da un mayor valor a los píxeles con mayor cambio de intensidad y, por tanto, se asociará un coste bajo ya que se trata de un borde. En caso de que no exista borde los valores serán menores.

- La **función dirección del gradiente** asocia un bajo coste si la dirección del enlace entre dos píxeles *p* y *q*, es perpendicular, o casi perpendicular, al gradiente en *p*, ya que esto sería propio de la existencia de frontera. Si la dirección del enlace es similar a la del gradiente asociará costes altos indicando que no hay fronteras.

- El **mapa de coste local** es la suma ponderada de las tres funciones anteriores. Entonces se define como

$$L(p,q) = w_Z f_Z(q) + w_G f_G(q) + w_D f_D(p,q)$$

donde $w_Z + w_G + w_D = 1$ y $w_Z, w_G, w_D \in \Re$

- El **algoritmo de Dijkstra** determina el camino más corto entre un nodo y el resto pertenecientes a un grafo.

- El **mapa de coste acumulado** se obtiene aplicando el algoritmo de Dijkstra al mapa de coste local.

- El principal problema encontrado durante la implementación es el **excesivo tiempo** empleado en realizar la segmentación. Para solucionarlo se han incluido dos **modificaciones** al método:

  - **Creación de subimágenes** más pequeñas a las que realizar todo el proceso computacional.

  - Aplicación de un **margen** elegido por el usuario para evitar los contornos no encuadrados.