Visión por Computador

Proyecto Final:
Implementación del Algoritmo de Seam Carving

Curso 2019-2020
Cuarto Curso del Grado en Ingeniería Informática

Paula Iglesias Ahualli 75574877M
piglesias@correo.ugr.es

Laura Rabadán Ortega 79088745W
laurarabadan@correo.ugr.es

CONTENIDO

1. INTRODUCCIÓN

El paper en el que basamos nuestro proyecto, *Seam Carving for Content-Aware Image Resizing*[1], publicado en 2007 en SIGGRAPH afronta el problema de modificar el tamaño de una imagen sin afectar al contenido. Cuando la imagen se quiere reducir, se intenta no modificar información que se considera relevante por deformación de la imagen. De la misma manera, al aumentar el tamaño, se quiere evitar *estirar* los objetos, deformarlos al buscar mayores tamaños. Con *Seam Carving* se intenta dar una solución a estos problemas.

Los autores, Avidan y Shamir, proponen el concepto de *seam* ("costura"), en el que se basa la solución. Consiste en un camino óptimo de píxeles conectados de manera vertical, de abajo hacia arriba de la imagen, u horizontal, de derecha a izquierda, con menor energía de la imagen, es decir, los píxeles de menor relevancia. Se eliminarán o añadirán los *seams* necesarios para conseguir el tamaño deseado.

[1] Seam carving for content-aware image resizing (Julio, 2007), S. Avidan, A. Shamir, SIGGRAPH '07: ACM SIGGRAPH

2. FUNCIONES DE ENERGÍA

Para conseguir el camino de píxeles de menor relevancia que se mencionó en el apartado anterior, se necesita calcular la energía de la imagen, para seleccionar la *seam* de menor energía.

La primera función para este propósito que se propone en el paper estudiado, *e1*, consiste en calcular la suma de la derivada de la imagen en *x* y en *y*.

$$e_1(I) = \left|\frac{\delta}{\delta x} I\right| + \left|\frac{\delta}{\delta y} I\right|$$

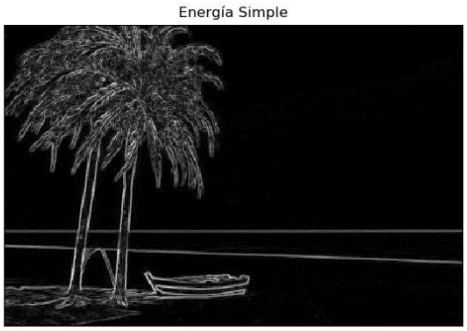
A la hora de implementarla, se utiliza la función de *Sobel* de OpenCV, calculando, por un lado, la derivada de la imagen en x y, por otro lado, la derivada en y.

```
def simpleEnergy (image):
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    image = image.astype(np.float)

    x = np.abs(cv2.Sobel(image, cv2.CV_64F, 1, 0))
    y = np.abs(cv2.Sobel(image, cv2.CV_64F, 0, 1))

return x + y
```

Como esta función trabaja con la imagen en formato monobanda, implementamos una segunda función de energía, similar a la anterior, pero trabajando con la imagen en formato tribanda. Esta vez calculando la suma de las derivadas para cada banda. El resultado es la suma de todas las derivadas conseguidas.

```
def simpleEnergyRGB(image):
    b, g, r = cv2.split(image)

    b_energy = np.absolute(cv2.Sobel(b, cv2.CV_64F, 1, 0, ksize=3)) +
               np.absolute(cv2.Sobel(b, cv2.CV_64F, 0, 1, ksize=3))
    g_energy = np.absolute(cv2.Sobel(g, cv2.CV_64F, 1, 0, ksize=3)) +
               np.absolute(cv2.Sobel(g, cv2.CV_64F, 0, 1, ksize=3))
    r_energy = np.absolute(cv2.Sobel(r, cv2.CV_64F, 1, 0, ksize=3)) +
               np.absolute(cv2.Sobel(r, cv2.CV_64F, 0, 1, ksize=3))

    return b_energy + g_energy + r_energy
```

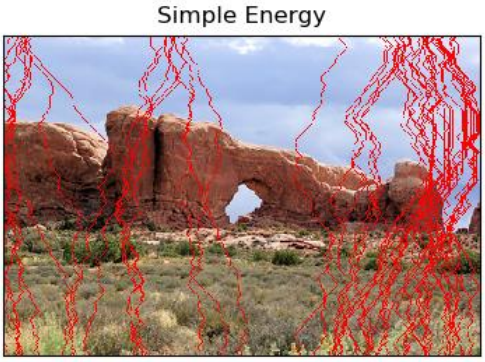
A continuación, se muestra un ejemplo de la salida de cada función de energía:

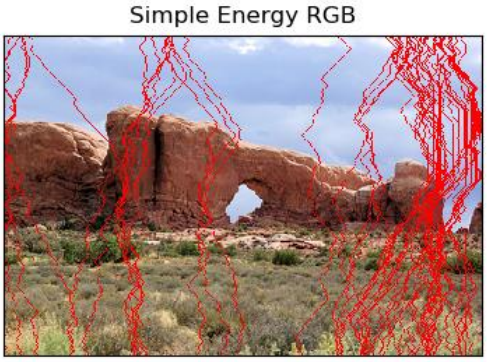


En las imágenes anteriores, dónde se muestra solo la energía, no se aprecian grandes diferencias entre las dos, salvo que los bordes de RGB tienen un pequeño "halo", en comparación con los resultados de la energía simple.

A continuación, observamos como haciendo una reducción simple con el algoritmo *Seam Carving* la imagen resultante es prácticamente idéntica, siendo difícil distinguir diferencias, incluso entre las *seams* dibujadas. Se comentan otras posibles funciones de energía, pero una de las que proporciona mejores resultados es e_1. No existe una función de energía que proporcione buenos resultados con cualquier imagen, por lo que, atendiendo a los resultados experimentales, implementaremos e_1 con ***Energy RGB.***

Hay que tener en cuenta que a la hora de pintar las *seams* se utilizan los píxeles que se van a eliminar, por lo que, a medida que avanzamos en el número de *seams* eliminadas, las representaciones pueden perder exactitud. Aún así, sirven para hacernos una idea aproximada del funcionamiento del algoritmo.





3. MODIFICAR UNA SEAM

La técnica comentada puede utilizarse para dos tareas básicas: reducir el tamaño de una imagen eliminando *seams*, y aumentarlo, duplicando los *seams*.

3.1. Eliminar una seam

Para disminuir el tamaño de una imagen, se deben eliminar *seams* de la imagen, seleccionándolos previamente. Para ello, se utiliza la energía de la imagen y se calcula la matriz de energía acumulativa.

La matriz de energía acumulativa, *M*, es esencial para este proceso. La idea es crear todos los caminos de menor energía acumulada y, a partir de ellos, elegir aquel que menos energía tenga. Para rellenar esta matriz, se empieza desde la segunda fila de la matriz de energía y se rellena la primera fila de *M* con 0. Desde este momento, cada píxel tendrá la energía correspondiente a esta función:

$$M(x,y) = e(x,y) + \min \begin{cases} M(x-1,y-1) \\ M(x,y-1) \\ M(x+1,y-1) \end{cases}$$

Dónde:

- $M(x,y)$ es el píxel que estamos calculando.
- $e(x,y)$ es la energía del píxel.
- $M(x-1,y-1)$ es el píxel superior izquierdo al que estamos calculando.
- $M(x,y-1)$ es el píxel superior al que estamos calculando.
- $M(x+1,y-1)$ es el píxel superior derecho al que estamos calculando

La idea es, de los posibles píxeles superiores a los que se puede ir, elegir el que mejor energía tenga. Como se mira en la matriz de energía acumulativa, esos píxeles posibles habrán elegido el padre con menor energía, por lo que vamos consiguiendo un camino de energía mínimo.

```
def Seam (image, energy):
    n, m = image.shape[:2]

    M = energy.copy()  # Matriz para la energía acumulativa mínima

    # Recorremos la imagen desde la segunda fila hasta la última
    for i in range (1, n):
        if m > 1:
            M[i,0] = energy[i,0] + min(M[i-1, 0], M[i-1,1])

        else:
            M[i,0] = energy[i,0] + M[i-1, 0]

        for j in range (1, m-1):
            M[i,j] = energy[i,j] + min(M[i-1,j-1], M[i-1,j], M[i-1,j+1])
```

```
        M[i,m-1] = energy[i,m-1] + min(M[i-1, m-2], M[i-1,m-1])

    return M
```

Una vez tenemos esta matriz rellena, buscamos en su última fila el menor valor. Desde este valor se buscará, de manera inversa a como hemos rellenado M, la *seam* que supone un menor coste de energía. Se guardan los índices que forman parte de la *seam* óptima para eliminarlos.

```
def crearCamino (M):
    n, m = M.shape[:2]

    camino = np.zeros((n), dtype=np.int)
    camino[0] = np.argmin(M[-1])

    # Camino de la costura, buscando la menor energía posible
    for i in range (1, n):
        indy = camino[i-1]
        min_ind = indy
        min_value = M[n - i - 1, indy]

        if (indy - 1) > -1 and min_value > M[n - i - 1, indy - 1]:
            min_ind = indy – 1
            min_value = M[n - i - 1, indy - 1]

        if (indy + 1) < m and min_value > M[n - i - 1, indy + 1]:
            min_ind = indy + 1
            min_value = M[n - i - 1, indy + 1]

        camino[i] = min_ind

    return camino
```

Este proceso es sencillo: simplemente, a partir del índice guardado, desplazamos toda la fila a la izquierda. Si el píxel es el último de su fila, no se hace nada. Hay que tener en cuenta que solo se va a eliminar una *seam* de cada fila, por lo que sobrará la última columna de la imagen, que se elimina.

```
def removeSeam (image, camino):
    n, m = image.shape[:2]

    for i in range (0, n):
        for j in range (camino[i], m - 1):
            image[n - i - 1, j] = image[n - i - 1, j + 1]

    return np.delete(image, -1, 1)
```

Si se quiere eliminar una *seam* horizontal, se utiliza el mismo procedimiento. Para evitar rehacer la función, utilizamos los métodos anteriores, pero rotando la imagen 90 grados, de manera que las filas pasan a ser las columnas y viceversa.

```
image = np.rot90(image, k=-1, axes=(0, 1))
image = np.rot90(image, k=1, axes=(0, 1))  # Para dejarla como antes
```

Cómo al eliminar una única *seam*, ya sea vertical u horizontal, la diferencia con respecto a la original no se aprecia, mostramos el resultado pintando las *seams* seleccionadas.

3.2. Añadir una seam

Para añadir una *seam* el procedimiento es similar al anterior. Debemos buscar, de la misma manera que antes, la *seam* que consiga un camino de menor energía, ayudándonos de la matriz M.

Esta vez, tenemos que duplicar la seam seleccionada con el valor promedio de sus vecinos de la izquierda y derecha. Cómo se quiere duplicar el píxel, para que quede acorde a la imagen, hay que modificar también el píxel que pertenece a la *seam*, para que no se note un salto brusco en la imagen.

Por tanto, suponemos que tenemos un píxel que pertenece a la *seam*. En ese punto tenemos que añadir dos nuevos píxeles: el primero haciendo el promedio entre el píxel original y su vecino de la izquierda, y el segundo con el promedio entre el píxel original y su vecino de la derecha. De esta manera, los dos píxeles guardan relación con el original, ya que uno de los valores que le influye es el del propio píxel original, pero cada uno está más relacionado con el píxel vecino. En el caso de los píxeles que son bordes, el del borde será el original y al otro se le hará el promedio.

```
def addSeam (image, camino):
    n, m = image.shape[:2]

    # Se crea una nueva imagen acorde a su tamaño

    for i in range (0, n):
        # Se copia la parte de la imagen que se mantiene intacta

        # Se comprueba si el píxel está en el borde, si está, no se hace el
        # promedio, se añade el nuevo pixel a la derecha, haciendo el
        # promedio con el vecino de la derecha
        if camino[i] > 0:
            left = image[n - i - 1, camino[i] - 1] *
                   image[n - i - 1, camino[i] - 1]

            center = image[n - i - 1, camino[i]] *
                     image[n - i - 1, camino[i]]

            img[n - i - 1, camino[i]] = # Promedio de los píxeles

        else:
            img[n - i - 1, camino[i]] = image[n - i - 1, camino[i]]

        # Se copia el resto de la imagen

    return img
```

Para hacer el promedio de dos píxeles no basta con sumar los dos valores y dividirlos entre dos, como se hace normalmente. En este caso, hay que multiplicar cada valor por si mismo antes de sumarlos y dividirlos entre dos. A este resultado le hacemos la raíz cuadrada y conseguimos el valor promedio de los dos píxeles.

$$Media = \sqrt{\frac{x^2 + y^2}{2}}$$

Al igual que en el caso anterior, para añadir una *seam* horizontal el procedimiento es el mismo.

4. MODIFICAR VARIAS SEAMS

Cuando queremos modificar el tamaño de una imagen, lo normal es que tengamos que añadir o eliminar más de una *seam*, ya sea horizontal o vertical. Para el caso de eliminar *seams*, basta con repetir de manera iterativa el proceso anterior tantas veces como filas o columnas quieran añadirse o eliminarse.

4.1. Eliminar varias seams

El problema surge cuando se quieren modificar las dos dimensiones de la imagen $\mathrm{I}(n, m)$. En este caso, los autores plantean la pregunta de en qué orden eliminar las *seams*. Como solución, proponen utilizar una matriz, T, de dimensiones $r \times c$, siendo $r = n - nn$ y $c = m - nm$, en la que se guarde, en cada casilla, el mínimo coste para conseguir ese tamaño. Es decir, si estamos en $T[1,4]$, el mínimo coste posible para reducir 1 fila y cuatro columnas de la imagen original será el valor que contenga.

Para rellenar la tabla, empezamos con $T[0,0]$ a 0, ya que el coste mínimo para conseguir el tamaño actual es nulo. El resto de las casillas se rellena calculando:

$$T(r,c) = \min \begin{cases} T(r-1,c) + E(s^X(I_{n-r-1\times m-c})) \\ T(r,c-1) + E(s^Y(I_{n-r\times m-c-1})) \end{cases}$$

Es decir, el mínimo coste entre la energía de la casilla a la izquierda más el coste de quitarle una *seam* vertical a la imagen actual y la energía de la casilla de encima más el coste de quitarle una *seam* horizontal a la imagen actual. En una matriz auxiliar con las mismas dimensiones que T vamos guardando la opción que se eligió en cada momento para cada casilla de T.

```
for j in range (0, c):
   for i in range (0, r):
      hor_min, hor_indx, hor_path = verticalSeam(image)
      vert_min, vert_min, vert_path = verticalSeam(image)

      T[i,j] = min(T[i-1,j] + hor_min, T[i, j-1] + vert_min)

      if T[i,j] == T[i, j-1] + vert_min:
         options[i,j] = 1

      image = removeSeam(image, hor_path)

   image = removeSeam(image, path)
```

Tras $r * c$ iteraciones, tendremos un mapa de menores energías para cada tamaño posible, eliminando un máximo de *r* filas y *c* columnas. La casilla $T[r,c]$ contendrá el mínimo coste de energía posible para conseguir el tamaño deseado.

A partir de los valores de la matriz T, seleccionamos el orden en el que se va a aplicar la reducción de *seams*. Empezando en $T[r,c]$, vamos eligiendo el vecino, izquierdo o superior (exclusivamente), con menor energía. Si hemos elegido el vecino de la izquierda, guardamos

como opción *seam* horizontal; si, por el contrario, elegimos el vecino superior, guardamos *seam* vertical. Repetimos este proceso hasta llegar a $T[0,0]$.

```
while cont < order.shape[0]:
   if options[r,c]:
      if c > -1:
      order[cont] = 1
         c--

      else:
         order[cont] = 0
         r--

   else:
      if r > -1:
      order[cont] = 0
         r--

      else:
         order[cont] = 1
         c--

   cont += 1
```

A la hora de implementarlo, nos dimos cuenta de que cada píxel acumula el menor coste de los dos píxeles entre los que se tendrá que elegir a la hora de buscar el orden. Por tanto, si en vez de la matriz T utilizamos la matriz en la que guardamos la información del tipo de *seam* eliminada en cada momento. De esta manera, si $T[r, c]$ es *seam* horizontal, le restamos 1 a r, ya que hemos eliminado una fila de la imagen; si, por otro lado, es *seam* vertical, le restamos 1 a c, ya que hemos eliminado una columna.

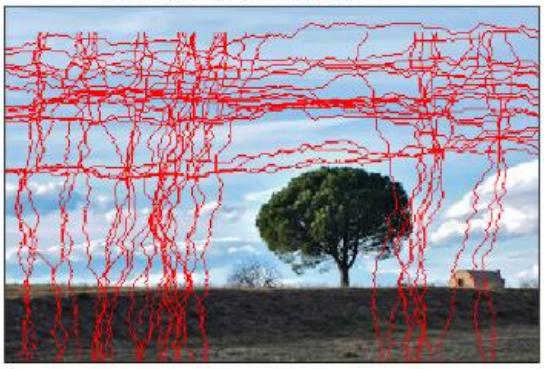
Una vez tenemos seleccionado el orden en el que vamos a ir eliminando *seams*, resultado de la función anterior, solo debe irse eliminando *seams* horizontales y verticales, según se indique, con el mismo proceso con el que se eliminaba una sola.

El resultado obtenido es bastante bueno. Si comparamos los resultados obtenidos con la imagen original (figura superior), apenas se aprecia diferencia, eliminando 20 filas y 35 columnas.

Reducción: Filas = 247Columnas = 365

Reducción eficiente: Filas = 247Columnas = 365

Reducción: Filas = 247Columnas = 365



Reducción eficiente: Filas = 247Columnas = 365

4.2. Añadir varias seams

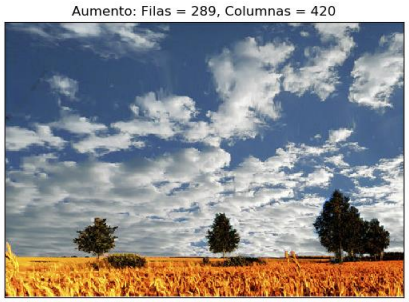
Si en vez de reducir la imagen queremos aumentarla, no basta con utilizar un bucle que vaya añadiendo *seams*, ya que todas tenderían a añadirse en el mismo lugar que es donde estaría la menor energía. Para evitar esto, que produciría una deformación en la imagen, se deben coger las *k* mejores *seams* para eliminar. De esta manera el aumento estará repartido por la imagen.

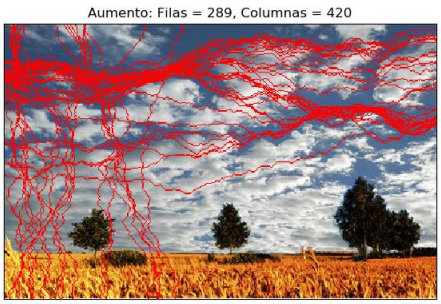
Para representar este comportamiento en nuestro programa, se realiza el procedimiento anterior (eliminar varias *seams*) y guardamos los caminos que se seleccionan para eliminar. Cuando se cambie de *seam* vertical a horizontal o viceversa, aumentamos las *seams* según los caminos seleccionados.

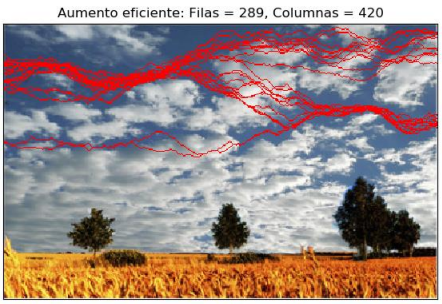
```
for o in order:
   if o != anterior:
      for i in range (len(caminos)):
         image = addSeam (image, caminos[i])

      orden = []
      caminos = []
      anterior = o
      aux = image.copy()

   orden.append(o)

   a, b, path = verticalSeam (aux, funcion)
   aux = removeSeam (aux, path)

   caminos.append(path)
```

Con este procedimiento se consigue un resultado más correcto al que se conseguiría al añadir *seams* de manera iterativa.

De nuevo, no se aprecian cambios significativos al añadir filas y columnas a la imagen. Lo más interesante es que el tamaño de los árboles se preserva y, como podemos ver, las regiones de baja energía se concentran en el cielo, como uno esperaría de forma intuitiva. Para este ejemplo se han añadido 35 filas y 20 columnas.







4.3. Añadir y eliminar varias seams

Si en vez de eliminar o aumentar tanto las filas como las columnas queremos aumentar una de las dos y disminuir la otra, dividimos el problema en dos: primero reducimos a imagen en la cantidad que se pide y después la aumentamos. Como una de las operaciones va a trabajar con las filas y la otra con las columnas, no interfieren la una con la otra.

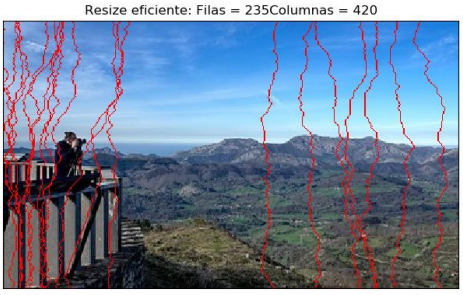
Realizamos primero la operación de eliminar para evitar cálculo innecesario: si aumentamos y después reducimos, eliminamos píxeles que hemos tenido que calcular. Este caso se contempla en la función que se encarga de reducir el tamaño de una imagen. Cuando una de las dimensiones se quiere reducir y la otra ampliar, primero llama a reducir, manteniendo la dimensión que se va a ampliar intacta, y después a aumentar, con el mismo criterio que antes.

```
def contentAwareResizing (img, nn, nm):
    n, m = img.shape[:2]
    dif_n = n - nn
    dif_m = m - nm

    if dif_n > 0:
        if dif_m < 0:
```

```
            img = funcion(img, nn, m, accion[0], energy, draw)
            img = funcion(img, nn, nm, accion[1], energy, draw)

        else:
            img = funcion(img, nn, nm, accion[0], energy, draw)

    elif dif_n < 0:
        // Espejo del caso anterior

    else:
        // Solo se eliminan horizontales o verticales

    return img
```

En primer lugar, disminuimos las filas y aumentamos las columnas, en 30 y 20 respectivamente. Como se explicó, el problema se divide en dos partes: primero reducimos y después ampliamos. Es por esto por lo que solo se pintan las *seams* verticales, las que añadimos.

Nuevamente, no se podría distinguir a simple vista cambios entre las imágenes.



Añadimos otro ejemplo, esta vez aumentando las filas y reduciendo las columnas, para comprobar que no afecta a los resultados finales. En todos los casos, obtenemos imágenes sin distorsión aparente.

Resize: Filas = 330Columnas = 380

Resize eficiente: Filas = 330Columnas = 380

Resize: Filas = 330Columnas = 380

Resize eficiente: Filas = 330Columnas = 380

5. Caso Práctico: Eliminar un Objeto

Una aplicación práctica de este método es la eliminación de objetos de una imagen. Si tenemos una imagen de la que queremos suprimir algún elemento, podemos aplicar este método para conseguir buenos resultados.

Para conseguir el propósito propuesto, necesitamos utilizar máscaras. Vamos a diferencias entre dos máscaras distintas: máscara negativa y máscara positiva. La máscara negativa es una imagen con fondo blanco en la que resaltamos en negro el elemento que queremos eliminar. Por el contrario, la máscara positiva tiene el mismo formato que la máscara negativa, pero en negro tenemos resaltado el objeto que queremos mantener.

Por ejemplo, de la imagen siguiente queremos eliminar el elemento del centro y mantener los dos de los lados, por lo que esas son sus máscaras negativas y positivas correspondientes.

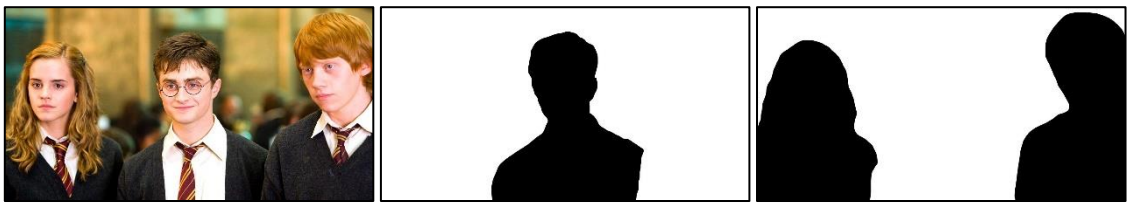


Imagen original *Máscara negativa* *Máscara positiva*

Con la máscara negativa calculamos el número de *seams* verticales u horizontales que tenemos que eliminar. Para ello creamos una función que nos calcula el máximo número de cada tipo que se tendría que eliminar, y escogemos el menor valor. Por ejemplo, en este caso, está claro que el número de filas será mucho mayor que el número de columnas, por eso, se elimina el objeto por columnas.

```
def maskSize(mask):
    rows, cols = np.where(mask < 200)
    height = np.amax(rows) - np.amin(rows) + 1
    width = np.amax(cols) - np.amin(cols) + 1

    return height, width
```

El resto del proceso es sencillo. Como en el resto de los casos, debemos calcular la energía de la imagen, pero, a diferencia de lo que se hacía en los casos anteriores, esa matriz de energía se ve influenciada por las máscaras negativa y positiva.

La máscara negativa va a cambiar la energía correspondiente al objeto que queremos eliminar, haciéndola muy pequeña. Es decir, todos los píxels que no tengan valor 255 (blanco) en la máscara cambiaran su valor por -100.

```
def removeEnergy(energy, mask):
    n, m = energy.shape[:2]

    for i in range (n):
        for j in range (m):
```

```
            if mask[i,j] < 255:
                energy[i,j] = -100

    return energy
```

Por otro lado, la máscara positiva va a aumentar la energía correspondiente al objeto que se quiere mantener, haciéndola muy grande. Todos los píxeles con valores menores a 255, que no sean blancos, se les aumenta la energía para evitar seleccionarlo como *seam*.

```
def preserveEnergy(energy, mask):
    n, m = energy.shape[:2]
    maxi = energy.max() + 1000

    for i in range (n):
        for j in range (m):
            if mask[i,j] < 255:
                energy[i,j] = maxi

    return energy
```

Con esta nueva matriz de energía, calculamos la *seam* óptima correspondiente, horizontal o vertical, y la eliminamos. Aumentamos este proceso hasta eliminar todas las *seams* que se habían calculado en el paso anterior.

Al aplicar las máscaras a la imagen anterior, se consigue el resultado siguiente:

Imagen original *Objeto eliminado*

A continuación, mostramos otros ejemplos con sus máscaras correspondientes, esta vez sin máscaras positivas.

En las imágenes siguientes intentamos replicar el ejemplo ilustrativo del paper en el que eliminaban un zapato. Resulta muy difícil encontrar el coche que no aparece en la imagen de la derecha.

En este caso, eliminamos una moto de la izquierda y aumentamos la imagen hasta su tamaño original. No se aprecian grandes distorsiones en el resultado.

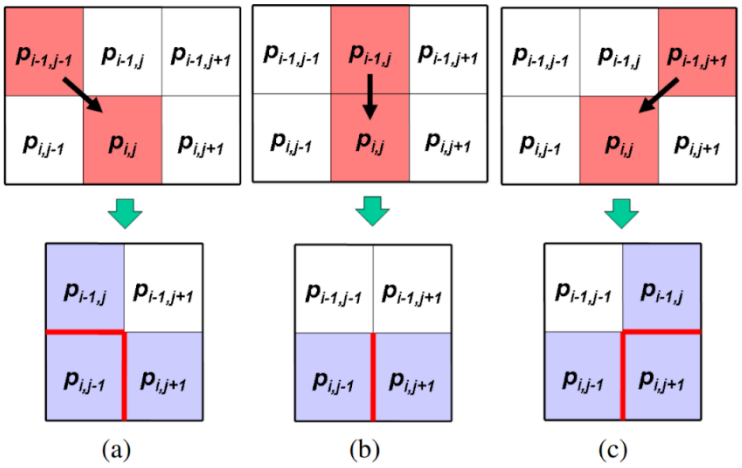
6. MEJORAS

6.1. Forward Energy

Un año más tarde, los mismos autores publicaron el paper "*Improved Seam Carving for Video Retargeting*[2]", donde el objetivo era aplicar *Seam Carving* a vídeos, pero aparte introdujeron una mejora respecto al algoritmo original que puede aplicarse en imágenes estáticas: *forward energy*, que resulta en una mejora muy significativa, como veremos a continuación.

Como su nombre indica, mejora la función de energía. Hasta ahora el criterio para eliminar las *seams* se basaba en escoger la de menor energía, pero no teníamos en cuenta la **energía global** que se añade a la imagen al eliminar una *seam*. Esto último sucede porque píxeles vecinos que antes no lo eran, ahora lo son, por lo que la energía global de la imagen aumenta (o disminuye).

Forward energy predice qué píxeles serán adyacentes y se basa en ello para elegir la mejor *seam* a eliminar:



Definimos D como la diferencia de color entre dos píxeles p_0 y p_1:

$$D[p_0, p_1] = D[(x_0, y_0), (x_1, y_1)]$$
$$D[p_0, p_1] = (R_{p_0} - R_{p1})^2 + (G_{p_0} - G_{p1})^2 + (B_{p_0} - B_{p1})^2$$

La energía asociada a un píxel tendrá tres costes distintos, atendiendo a los tres casos que pueden darse al eliminar un píxel: si este está conectado a una *seam* arriba a la izquierda, a una *seam* justo arriba, o a una *seam* arriba a la derecha, tal y como se muestra en la imagen.

Los costes serán en cada caso:

[2] Improved Seam Carving for Video Retargeting (2008), M. Rubinstein, A. Shamir, S. Avidan, ACM Transactions on Graphics (SIGGRAPH), 27, 3

$$C_L(x,y) = D[(x-1,y),(x+1,y)] + D[(x,y-1),(x-1,y)]$$
$$C_U(x,y) = D[(x-1,y),(x+1,y)]$$
$$C_R(x,y) = D[(x-1,y),(x+1,y)] + D[(x,y-1),(x+1,y)]$$

Ecuación 1 - Función de costes

Podemos ver como el primer coste $D[(x-1,y),(x+1,y)]$ aparece en los tres casos, ya que los píxeles a los lados del píxel a eliminar se convierten siempre en vecinos.

Para resolver este problema volvemos a aplicar programación dinámica.

A cada píxel le asociaremos la energía de la energía mínima final de la *seam* que **acaba** en ese píxel. Para esto nos fijaremos en cada subproblema en las *seams* de los píxeles arriba a la izquierda, arriba en el centro y arriba a la derecha del píxel en cuestión.

Para escoger por cuál de estos píxeles continuamos, escogeremos el mínimo de los siguientes costes:

$$M(x,y) = \min \begin{cases} M(x-1,y-1) + C_L(x,y) \\ M(x,y-1) + C_U(x,y) \\ M(x+1,y-1) + C_R(x,y) \end{cases}$$

Como es natural, el paper no explica los casos particulares de las esquinas y bordes de la imagen. Nosotras hemos aprovechado la función de crear un borde con OpenCV para crear un borde que replique los píxeles (cv2.BORDER_REPLICATE) creando bordes superior, izquierdo y derecho.

```
img = cv2.copyMakeBorder(img, top=1, bottom=0, left=1, right=1
                         borderType=cv2.BORDER_REPLICATE)
```

Con ayuda de `np.roll` calculamos la fila superior, y la columna izquierda y derechas para cada píxel de la imagen con bordes.

```
U = np.roll(img, 1, axis=0)
L = np.roll(img, 1, axis=1)
R = np.roll(img, -1, axis=1)
```

Procedemos a calcular los costes, siguiendo la *ecuación 1*:

```
cU = np.abs(R - L)
cL = np.abs(U - L) + cU
cR = np.abs(U - R) + cU
```

Y finalmente eliminamos los bordes con `slicing`, por lo que no hemos tenido que tratar las esquinas de forma particular.

```
cU = cU[1:, 1:-1]
cL = cL[1:, 1:-1]
cR = cR[1:, 1:-1]
```

$M[0]$ será igual a $cU[0]$, el caso en el que no hay *seams* en la fila anterior.

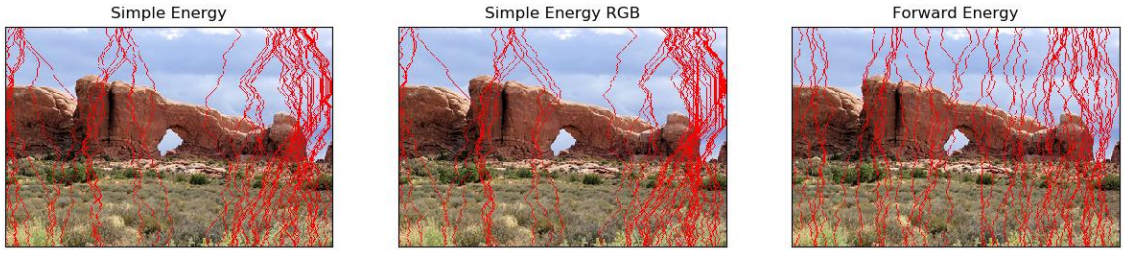
Ahora para cada fila empezando en 1 calculamos tres matrices mU, mL y mR volviendo a utilizar `np.roll`:

```
mU = M[i-1]
mL = np.roll(mU, 1)
mR = np.roll(mU, -1)
```

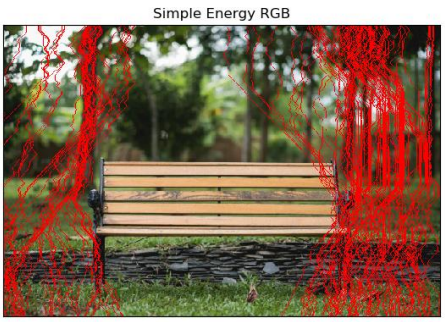
Finalmente sumamos los costes como se especifica en la función y escogemos el mínimo coste:

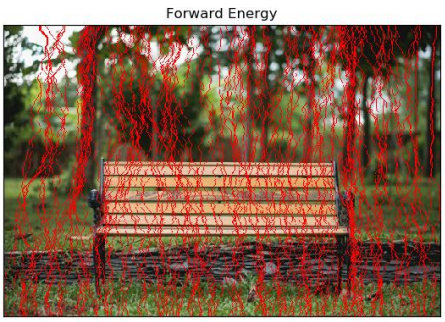
```
mLUR = np.array([mL, mU, mR])
cLUR = np.array([cL[i], cU[i], cR[i]])
mLUR += cLUR

argmins = np.argmin(mLUR, axis=0)
M[i] = np.choose(argmins, mLUR)
energy[i] = np.choose(argmins, cLUR)
```

Ya tenemos nuestra matriz *energy*, que tendrá para cada píxel la energía mínima final de la *seam* que **acaba** en ese píxel.



Si comparamos las tres energías utilizadas, se ve una clara diferencia entre l *Forward Energy* con respecto a las otras dos. Una ventaja interesante de este método de energía es que, al tener en cuenta la "energía futura" de la imagen resultante, las *seam* estarán repartidas por la imagen, por lo que sus objetos se distorsionan.





Si eliminamos una gran cantidad de columnas utilizando *Simple Energy RGB* y *Forward Energy*, el resultado que se consigue con el segundo método guarda más relación con la imagen original. La distancia del objeto con los bordes y el tamaño del banco con respecto a la imagen son prueba de ello.

Este método resulta muy interesante porque también elimina partes que podríamos considerar importantes, como píxeles del banco, pero que, al ver los resultados, estos son mejores.

6.2. Scale

Otra mejora interesante es la propuesta por Beuntorki en mayo de 2016. Propone redimensionar una imagen con el método *Seam Carving* una sola vez. Es decir, si queremos modificar el alto y ancho de una imagen, solo calcularíamos las *seams* verticales u horizontales, pero no ambas. Esta modificación reduce considerablemente el tiempo de ejecución.

Consiste en reescalar la imagen y modificar las *seams* que hacen falta para conseguir el tamaño objetivo. El reescalado lo marca el máximo valor, en el caso de reducir tamaño, o el mínimo valor, en el caso de aumentar la imagen, de las escalas, como se muestra a continuación:

Con esto conseguimos uno de los tamaños correctos, de la altura o la anchura, por lo que solo falta ajustar la otra dimensión, utilizando el método de *Seam Carving*.

Como se demuestra con los ejemplos a lo largo de esta memoria, los resultados de ambos métodos, sin el escalado y con él, son prácticamente iguales.

```
def scaleAndCarve (img, nn, nm, accion=removeOrderSeams):
    n, m = img.shape[:2]

    if accion == removeOrderSeams: scale_factor = max(nn/n, nm/m)
    else: scale_factor = min(nn/n, nm/m)

    height = int(n * scale_factor)
    width = int (m * scale_factor)
    dim = (width,height)

    # resize image
    resized = cv2.resize(img, dim)

    # Rotamos
    if abs(height - nn) != 0:
        order = np.ones((abs(height - nn)))

    elif abs(width - nm) != 0:
        order = np.ones((abs(width - nm)))

    else:    # Si no hace falta ajustar nada
        return resized

    # Eliminamos las verticales o horizontales que sobren
    resized = accion(resized, order)

    return resized
```

A lo largo de la memoria se han mostrado ambos resultados, con el método original (imágenes izquierdas) y con esta mejora (imágenes derechas). Como se puede comprobar, los resultados son equivalentes.

7. Conclusión y Casos críticos

Este método es muy interesante, pero no funciona con todas las imágenes.

Como se puede intuir, por las imágenes mostradas a lo largo de la memoria, devuelve buenos resultados con imágenes de paisajes, con zonas lisas, en las que no haya mucho detalle. Si probamos a utilizarlo con imágenes que contienen mucha geometría o probamos a eliminar un objeto muy relevante de la imagen, los resultados no son satisfactorios:

Las líneas rectas dejan de serlo al reducir la imagen: no hay ningún camino posible por el que puedan pasar las *seams* en el que no trastoquen la geometría de la imagen. Es por esto por lo que podemos observar líneas curvas acorde a los edificios.

En el caso de *object removal*, podemos ver que, al eliminar a la chica del medio, las otras caras se deforman al no haber suficiente espacio entre ellas, donde eliminar *seams*.

Una posible mejora para este problema en concreto podría ser combinar *seam carving* con el algoritmo de detección de caras (*face detection*) al igual que sugiere el paper.

Suponiendo una imagen a reescalar de dimensiones N x M, el algoritmo tiene una complejidad en tiempo de O(NxM+N+M) debido a la programación dinámica que implica. Esta alta complejidad supone un coste computacional considerable si usamos imágenes de alta resolución. Por este motivo consideramos que la mejora propuesta de utilizar el escalado antes de proceder con el algoritmo resulta muy significativa en el ahorro de tiempo de cómputo.

8. Referencias

La bibliografía base del proyecto ha sido el paper propuesto “Seam Carving for Content-Aware Image Resizing” citado al principio del documento.

Adicionalmente se ha estudiado la mejora propuesta por los mismos autores en el paper “Improved Seam Carving for Video Retargeting” también ya citado.

Como se tratan de trabajos altamente valorados en el campo de la Visión por Computador, hay mucho estado del arte al respecto. La bibliografía adicional que hemos consultado se muestra a continuación:

Real-world Dynamic programming: seam carving, A. Das, 29 Mayo 2019, https://medium.com/swlh/real-world-dynamic-programming-seam-carving-9d11c5b0bfca

Seam Carving, A. Campbell, 3 Diciembre 2018, https://andrewdcampbell.github.io/seam-carving

Content Aware Image Resizing Using Seam Carving, A. Bentourki, 3 Mayo 2016, http://www.aui.ma/sse-capstone-repository/pdf/spring2016/Content%20Aware%20Image%20Resizing%20Using.pdf

Seam Carving, X. Tao, Brown Education, Otoño 2018, http://cs.brown.edu/courses/cs129/results/proj3/taox/