Práctica 2 de Algorítmica

Seam carving (3 sesiones)

Curso 2015-2016

Seam carving

Seam carving

Seam carving es una técnica para reducir el tamaño de imágenes que es alternativa a un escalado. La idea es muy sencilla y se basa en eliminar estrechas franjas (de un pixel) que pasen por píxeles *poco importantes* de la imagen. Esta técnica se puede aplicar tanto en horizontal como en vertical.

Los pasos básicos son:

1. Calcular la *energía* o *gradiente* de cada pixel de la imagen.
2. Calcular los caminos más cortos o *seams*
3. Eliminar los pixels de esos caminos.

Puedes consultar en wikipedia para más información:

http://en.wikipedia.org/wiki/Seam_carving

Aunque los *seams* se pueden calcular usando diversas técnicas (por ejemplo, algoritmos de tipo flujo máximo/cortadura mínima), en esta práctica vamos a utilizar **programación dinámica**. En particular, se trata de encontrar el camino más corto en un grafo acíclico.

Atención

Sin pérdida de generalidad vamos a reducir el **ancho** de las imágenes eliminando columnas. Todo esto se podría aplicar de manera muy similar para reducir la altura.

Detalles de implementación

Detalles de implementación

En este apartado vamos a explicar la implementación de *seam carving* en python, las bibliotecas utilizadas y la estructura que os proporcionamos para realizar la práctica.

Dos notaciones para matrices

En el lenguaje python existen al menos dos formas diferentes y alternativas para representar matrices:

- Mediante un diccionario indexado por tuplas `(fila,columna)`.

```
m1 = {(i,j):0 for j in range(numcolumnas) for i in range(numfilas)}
m1[0,0] = 5 # azucar sintactico para m1[(0,0)] = 5
```

- Mediante una lista de listas, que se accedería con `[fila][columna]`.

```
m2 = [[0 for j in range(numcolumnas)] for i in range(numfilas)]
m2[2][1] = 5 # m[2] es una lista python correspondiente a la 3a fila
```

Además de estas dos formas, algunos objetos permiten acceder a sus elementos como si se trataran de matrices usando una notación u otra. Dos ejemplos son:

- Los arrays de la biblioteca `numpy` se acceden de manera `[x][y]` (en general pueden tener más de dos dimensiones).
- Los pixels de una `Image` de la biblioteca PIL se pueden indexar como `[x,y]`

Indexando imágenes y matrices

Para *marear* un poco más, os recordamos que la nomenclatura comunmente utilizada para especificar las coordenadas de una imagen difiere de la nomenclatura utilizada en matemáticas para indexar matrices.

- En matemáticas se indexa con `[fila,columna]` de modo que las filas empiezan en 1 desde arriba y se incrementan a medida que bajamos. Las columnas empiezan en 1 desde la izquierda y sus índices crecen hacia la derecha.
- Las imágenes se suelen indexar como `[x,y]` siendo `x` la coordenada horizontal (la columna) y siendo `y` la coordenada vertical (fila) que crece de abajo hacia arriba (al revés). Ambas coordenadas se indexan normalmente desde 0.

Advertencia

- En esta práctica vamos a utilizar las coordenadas `x` e `y` para referirnos a los ejes *horizontal* y *vertical* de la imagen, respectivamente. Esos índices irán desde `0` hasta `width-1` y hasta `height-1` respectivamente.
- **PERO** vamos a utilizar la notación de matrices (fila y columna) debido a que nos interesa representar las imágenes mediante una **lista de filas**. Por ello, para acceder a un pixel usaremos `[y][x]`.

Carga y manipulación de imágenes con PIL

PIL es la abreviatura de Python Imaging Library. Se trata de una biblioteca para el lenguaje Python que permite manipular imágenes así como cargarlas y salvarlas en varios formatos. Tienes un tutorial en este enlace

Un ejemplo de carga de una imagen:

```
from PIL import Image
import sys
file_name = sys.argv[1] # 1er argumento linea de comandos
img = Image.open(file_name)
max_x, max_y = img.size
```

Es interesante el comportamiento del método `load()` que devuelve un objeto que permite acceder a los pixels como si fuese un array bidimensional al estilo:

```
pix = img.load()
print pix[x, y]
pix[x, y] = value
```

Lo cual resulta mucho más rápido que con métodos tipo `getpixel` o `putpixel`. No obstante, en esta práctica vamos a convertir las imágenes a una **lista de listas** como se ha explicado anteriormente.

Carga y manipulación de imágenes con PIL

No obstante, lo que vamos a hacer es leer la imagen y convertirla en una lista de listas. Para ello procederemos en 2 etapas:

1. Convertirla en una matriz de la biblioteca numpy:

```
color_img = Image.open(file_name)
width,height = color_img.size
# convert the color image to a numpy array
color_numpy = numpy.array(color_img.getdata()).reshape(height, width,3)
```

2. Generar una lista de listas a partir del array numpy:

```
color_matrix = color_numpy.tolist()
```

Atención

- El motivo de hacer 2 etapas es que PIL y numpy tienen formas sencillas de *hablar entre sí*. En este caso numpy se está usando como una mera herramienta auxiliar que se podría eliminar a costa de utilizar bucles en python para ir leyendo los pixels de uno en uno, lo cual seguramente resultaría más lento.
- La imagen que cargamos está en color y por tanto cada pixel es una tripleta RGB, con lo que `color_matrix` es en realidad una lista de listas de listas (las últimas o más internas son las tripletas RGB, pero puedes ignorarlo/obviarlo).

Carga y manipulación de imágenes con PIL

- Para generar una imagen a partir de la matriz:

```
def matrix_to_color_image(color_matrix):
    return Image.fromarray(numpy.array(color_matrix, dtype=numpy.uint8))
```

- Luego es fácil salvarla a disco:

```
def save_matrix_as_color_image(color_matrix,filename):
    img = matrix_to_color_image(color_matrix)
    img.save(filename)
```

- También es conveniente generar una versión de la imagen en escala de grises y con valores de tipo `float`. De nuevo lo convertiremos en una lista de listas:

```
grayscale_img = color_img.convert("F")
grayscale_numpy = numpy.array(grayscale_img.getdata()).reshape(height,width)
grayscale_matrix = grayscale_numpy.tolist()
```

- Finalmente, crearemos una lista de listas similar pero para guardar el gradiente de la imagen en escala de grises:

```
gradient_matrix = [[0.0 for x in range(width)] for y in range(height)]
```

  Estas listas de listas contienen ambas valores de tipo `float`.

Ventajas de las listas de listas

¿Por qué tanto hincapié en representar las matrices mediante una lista de listas?

Muy sencillo: porque para eliminar un *seam* de la imagen hay que eliminar un pixel de cada fila de la imagen y éste pixel se encuentra en posiciones diferentes para cada fila.

La ventaja de usar una lista de listas es que cada fila es una lista y las listas tienen un método que permite eliminar un elemento de manera muy sencilla, tal y como se observa en el siguiente ejemplo:

```
>>> a = range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> a.pop(3)
>>> a
[0, 1, 2, 4, 5, 6, 7, 8, 9]
```

Tip

Que sea sencillo **no** quiere decir que sea eficiente, pues internamente las listas puede que se representen mediante vectores redimensionables y el coste es $O(n)$ como se ve en la siguiente página:

https://wiki.python.org/moin/TimeComplexity

Ventajas de las listas de listas

De este modo, pintar un seam de un determinado color (para temas de depuración) resulta trivial:

```
def paint_seam(height,seam_path,color_matrix,path_color=[0,0,0]):
    for y in range(height):
        color_matrix[y][seam_path[y]] = path_color
```

Y eliminar el seam, que es lo que más nos interesa, es igualmente sencillo:

```
def remove_seam(height,seam_path,color_matrix):
    for y in range(height):
        color_matrix[y].pop(seam_path[y])
```

Inconvenientes de las listas de listas

Un pequeño inconveniente es que hay que pasar las listas de listas a imagen. Esto no es problemático si guardamos la imagen resultante final, pero hace más lenta la animación.

Cosas a manipular

Para los ejercicios de esta práctica vamos a necesitar manipular los siguientes objetos de tipo *lista de listas*:

- La imagen original en color
- La imagen en escala de grises para calcular el gradiente
- La matriz que contiene el gradiente
- La matriz para los estados de programación dinámica

Adicionalmente, los caminos o *seams* obtenidos se pueden representar directamente mediante una lista de las columnas de cada fila.

Programa principal

Para facilitaros el desarrollo de esta práctica os proporcionamos un esqueleto del programa. Este programa recibe argumentos por la línea de comandos como se muestra en este fragmento:

```
if __name__ == "__main__":
    if len(sys.argv) < 4:
        print '\n%s image_file gradient_type{0|1|2} {num_column|%%}\n' \
              % (sys.argv[0],)
        sys.exit()
    file_name = sys.argv[1]
    gradient_type = int(sys.argv[2])
    ncolumns = sys.argv[3]
    ...
```

La versión inicial del programa tiene las siguientes características:

- El cálculo del *seam* está reemplazado por una función que devuelve un camino aleatorio.
- Solamente implementa un tipo de cálculo del *gradiente* de la imagen.
- Para eliminar varias columnas, el algoritmo procede eliminándolas **de una en una**. Esta técnica es mejor que otras pero resulta más lenta.

Animación del seam carving

Al ejecutar el programa como se muestra a continuación:

```
python seam_carving_pract.py BroadwayTowerSeamCarvingA.png 0 50%
```

Actividades a realizar

Actividades a realizar

- Completa la función que calcula el camino más corto que atraviesa todas las filas de la imagen.
  - Un camino válido puede empezar, terminar y pasar por cualquiera de las columnas donde esté definido el gradiente (todas menos la primera y la última) y solamente se puede mover en vertical o en diagonal. Es decir, cada vez que subes o bajas una fila, la coordenada de la columna solamente puede cambiar en -1,0 o 1.
  - Debes retornar uno de los caminos de menor coste. Para ello has de recuperar uno de los mejores caminos (podría haber empates) **sin utilizar backpointers**. El camino resultante se devolverá como una lista python de talla número de filas (valor `height`) que contiene el índice de la columna en cada posición. Es decir, los pixels que forman el camino son los de coordenadas de la imagen `(path[i],i)` si usamos el convenio de poner las coordenadas `(x,y)` siendo `x` la horizontal e `y` la vertical. Al acceder a una matriz representada mediante listas de listas sería la coordenada `[path[i]][i]`.

Actividades a realizar

- Completa la función `gradiente` para que sea posible elegir entre al menos 3 tipos de gradiente:
  1. El que se proporciona ya en la práctica
  2. El operador de Sobel, ver este enlace `http://en.wikipedia.org/wiki/Sobel_operator`
  3. La siguiente expresión `abs(m[y][x-1]-m[y][x+1])+abs(m[y-1][x]-m[y+1][x])`
- Haz que el programa pueda calcular el gradiente de manera **incremental**. Es decir, reutilizando las casillas de la matriz de gradientes de la iteración anterior que sigan siendo válidas y calculando únicamente las que habrán cambiado al eliminar un *seam* de la imagen.
- Haz una versión de programación dinámica que sea **incremental**. En este caso se trata de reutilizar los scores de la matriz de programación dinámica que consideres que es posible reutilizar. Ten en cuenta que si eliminamos un camino dejarán de ser válidos *"potencialmente"*:
  - Todos los scores que provengan de algunas de las casillas de ese camino.
  - Todos los scores cuyo gradiente haya podido cambiar de la iteración anterior a la actual.

Actividades a realizar

- Haz una versión donde el algoritmo de programación dinámica calcule varios seams simultáneamente. Para ello, has de tener en cuenta que una vez saques un camino éste no podrá utilizar pixels de cualquier camino anterior. En caso de encontrar un camino que solape con algún camino anterior debes descartarlo y pasar a otro. El número de seams a extraer depende de un número máximo `N` y de un porcentaje sobre el score del mejor camino (ejemplo: los caminos que sean hasta 95% tan buenos como el mejor).

Tip

Esta versión requiere una versión de la función `remove_seam` capaz de recibir una lista de `seam_path` y debes de borrar las columnas de cada fila de manera correcta (pista: de derecha a izquierda).