Visión por Computador (2014-2015)
Grado en Ingeniería Informática
Universidad de Granada

Proyecto Final: Intelligent Scissors

Alberto Quesada y Javier Moreno

11 de febrero de 2015

Índice

Índice de figuras

1. Introducción

En la actualidad el retoque fotográfico es una práctica cada vez más común tanto entre usuarios experimentados como amateurs. Uno de los requisitos básicos que se le suele pedir a las herramientas de edición de imagen (como Photoshop, GIMP, Pixelmator...) es la capacidad para extraer objetos de una escena. Esta tiene diferentes nombres dependiendo del software usado: Magic Wand Tool (Pixelmator), Lazo (Photoshop) o Tijeras (GIMP).

La delimitación de objetos manual es muy imprecisa y laboriosa. Aunque hay varias técnicas para automatizar este proceso, el problema sigue estando sin resolver por completo. La técnica propuesta para este proyecto se llama Intelligent Scissors (tijeras inteligentes) y permite extraer objetos de imágenes de forma rápida y bastante acertada.

El programa desarrollado sigue la técnica descrita en el artículo mort-sigg95[1], permitiendo al usuario utilizarla mediante una sencilla interfaz gráfica. No es necesario tener ningún conocimiento previo de la técnica o de ningún concepto de visión por computador.

Una vez seleccionada la imagen que se desea usar y elegido el objeto que se desea extraer, se tiene que ir pulsando con el ratón sobre varios puntos que delimiten el objeto y el software buscará la el camino óptimo para segmentación entre esos puntos.

2. Descripción del algoritmo

El algoritmo está basado en programación dinámica y formulado como un problema de búsqueda en grafo. La idea básica es encontrar el camino óptimo para segmentación entre dos píxeles definidos manualmente. Este camino óptimo es creado entre los dos píxeles y sus 8 vecinos, para encontrarlo se usa una función de coste entre un pixel y uno de sus vecinos.

$$l(\boldsymbol{p}, \boldsymbol{q}) = \omega_Z \cdot f_Z(\boldsymbol{q}) + \omega_D \cdot f_D(\boldsymbol{p}, \boldsymbol{q}) + \omega_G \cdot f_G(\boldsymbol{q})$$

Figura 2.1: Función de coste local

$$f_Z(\boldsymbol{q}) = \begin{cases} 0; & \text{if } I_L(\boldsymbol{q}) = 0 \\ 1; & \text{if } I_L(\boldsymbol{q}) \neq 0 \end{cases}$$

Figura 2.2: Cruzando por cero la laplaciana

[1] http://courses.cs.washington.edu/courses/cse455/02wi/readings/mort-sigg95.pdf

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

Figura 2.3: Magnitud del gradiente

$$f_D(\boldsymbol{p}, \boldsymbol{q}) = \frac{1}{\pi}\left\{\cos\left[d_{\boldsymbol{p}}(p,q)\right]^{-1} + \cos\left[d_{\boldsymbol{q}}(p,q)\right]^{-1}\right\}$$

Figura 2.4: Dirección del gradiente

$$d_{\boldsymbol{p}}(p,q) = \boldsymbol{D}'(\boldsymbol{p}) \cdot \boldsymbol{L}(\boldsymbol{p},\boldsymbol{q})$$
$$d_{\boldsymbol{q}}(p,q) = \boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) \cdot \boldsymbol{D}'(\boldsymbol{q})$$

Figura 2.5: Especificación de la dirección del gradiente

$$\boldsymbol{L}(\boldsymbol{p},\boldsymbol{q}) = \begin{cases} \boldsymbol{q}-\boldsymbol{p}; & \text{if } \boldsymbol{D}'(\boldsymbol{p}) \cdot (\boldsymbol{q}-\boldsymbol{p}) \geq 0 \\ \boldsymbol{p}-\boldsymbol{q}; & \text{if } \boldsymbol{D}'(\boldsymbol{p}) \cdot (\boldsymbol{q}-\boldsymbol{p}) < 0 \end{cases}$$

Figura 2.6: Link bidireccional entre los pixeles p y q

- fz: Cruzando por cero la laplaciana.
- fg: Magnitud del gradiente.
- fd: Dirección del gradiente.
- Wz, wd, wg son valores ya definidos que fueron calculados empiricamente por los autores del algoritmo.

El algoritmo principal recibe un pixel, teniendo el grafo de costes (pudiendo estar sin calcular algunos de sus valores al estar trabajando con programación dinámica), y mediante una lista de píxeles activos ordenados mediante sus costes, se van recorriendo sus vecinos y comprobando si pertenecen al camino óptimo o no.

3. Implementación

Para la implementación se han usado dos clases principales Window e IntelligentScissors, la primera maneja la presentación de imágenes y la segunda gestiona el algoritmo.

Se ha creado una estructura llamada Pixel para gestionar el grafo de búsqueda.

```
struct Pixel
{
        bool calculated;
        double N[8];
        double cost;
        Point point;
};
```

3.1. Clase Window

Window::Window(string filePath, string windowName)
Constructor de la clase Window que recibe la imagen a mostrar y el nombre de la ventana.

void Window::drawImage()
Método que dibuja la imagen en pantalla.

void Window::refreshImage()
Método que actualiza la imagen en pantalla cuando ha tenido alguna modificación.

void Window::touch(int event, int x, int y)
Método que controla los eventos de los clicks del ratón en la ventana de la imagen, y al seleccionar un punto lo pasa al algoritmo para construir el camino óptimo.

void Window::start()
Inicio de la interfaz.

3.2. Clase IntelligentScissors

IntelligentScissors::IntelligentScissors(string filePath, float wZ, float wD, float wG)

Constructor de la clase IntelligentScissors que recibe la imagen a procesar. Recibe la imagen, la lee y la convierte a escala de grises para poder trabajar con ella.

void IntelligentScissors::initData()
Método que inicializa las estructuras de datos a usar. Elimina el ruido de la imagen mediante un filtro de Gauss, calcula la Laplaciana, los gradientes, la matriz G e

inicializa las estructuras de datos necesarias para el algoritmo, entre ellas el grafo de costes.

void IntelligentScissors::calculateLocalCost(Point p)
Método que construye un pixel en el punto p y calcula el coste de sus 8 vecinos. Con este método es como se va rellenando el grafo de costes, ya que es llamado cuando un pixel tiene su variable booleana de calculado a false.

Pixel* IntelligentScissors::getPixel(Point p)
Devuelve un puntero a un pixel, si no está inicialado, lo inicializa. Este método es el que llamamos cuando necesitamos un pixel, o su coste entre el y uno de sus vecinos, lo que sería la función de costes del algoritmo: l(q, r).

bool IntelligentScissors::checkDimensions(Point p)
Comprueba si estamos accediendo a un punto fuera de las dimensiones de la imagen.

double IntelligentScissors::fZ(Point q)
Calcula el valor de fZ para el grafo de costes.

double IntelligentScissors::fD(Point p, Point q)
Calcula el valor de fD para el grafo de costes.

double IntelligentScissors::fG(Point q, int n)
Calcula el valor de fD para el grafo de costes.

void IntelligentScissors::NormalizePoint(Point p)
Normaliza un punto.

void IntelligentScissors::NormalizeSmallValues(double v)
Comprueba si un valor es demasiado pequeño, si es así lo iguala a 0.

Point IntelligentScissors::getNeighbor(Point p, int n)
Devuelve el n vecino de un punto p.

bool IntelligentScissors::isDiagonal(int n)
Comprueba si el vecino n está en la diagonal de un punto.

void IntelligentScissors::setError(int error)
Devuelve un error y termina el programa.

void IntelligentScissors::setPoint(int x, int y)
Define el punto siguiente en el algoritmo, lo pinta y pinta el camino óptimo que encuentra. Este es el método inicial del algoritmo.

void IntelligentScissors::drawPath(Point s)
Dibuja el camino encontrado entre dos puntos.

void IntelligentScissors::drawPoint(Point s)
Dibuja un punto en la imagen.

void IntelligentScissors::optimalPath(Point s)
: Algoritmo Live-Wire 2-D DP graph search. Calcula el camino óptimo entre dos puntos para segmentación.

bool IntelligentScissors::find(multiset<Pixel*, comp>L, Pixel* p)
: Busca si un Pixel pertecene a un multiset de Pixel.

const Mat IntelligentScissors::getImage()
: Devuelve la imagen.

vector<Point>IntelligentScissors::getPath()
: Devuelve el camino óptimo encontrado.

Como comentario a la implementación, para mejorar la eficiencia se ha usado una estructura de datos multiset para la lista de píxeles activos ordenada por sus costes. También como mejora de eficiencia se han usado punteros a píxeles en casi todos los cálculos evitando la construcción de píxeles y pudiendo modificarlos en el grafo de costes directamente.

3.3. Algoritmo principal:

```
void IntelligentScissors::optimalPath(Point s)
{
        // Pointers from each pixel indicating the minimum cost path.
        P = vector<vector<Point> >(imgGray.rows, vector<Point>(
        imgGray.cols));

        // Boolean function indicating if q has been
        // expanded/processed.
        vector<vector<bool> > e (imgGray.rows, vector<bool>(
        imgGray.cols, false));

        // List of active pixels sorted by total cost (initially empty).
        multiset<Pixel*, comp> L;

        Pixel *q, *r;
        double g_tmp;

        // Initialize active list with zero cost seed pixel.
        getPixel(s)->cost = 0.0;
        L.insert(getPixel(s));

        while(!L.empty()) // While still points to expand.
        {
                // Remove minimum cost pixel q from active list.
                q = *L.begin(); L.erase(L.begin());
                e[q->point.x][q->point.y] = true; // Mark q as expanded.
                for(int n=0; n<8; n++)
                {
                        Point a = getNeighbor(q->point, n);
                        r = getPixel(a);
                        if(r != NULL && !e[r->point.x][r->point.y])
                        {
                                // Compute total cost to neighbor.
                                g_tmp = q->cost + q->N[n];

                                if(find(L, r))
                                {
                                  if(g_tmp < r->cost)
                                // Remove higher cost neighbor's from list.
                                    L.erase(r);
                                }
                                else // if neighbor not on list.
```

```
                                        {
                                         // assign neighbor's total cost.
                                          r->cost = g_tmp;
                                         // set back pointer.
                                          P[r->point.x][r->point.y] = q->point;
                                         // and place on (or return to) active list.
                                          L.insert(r);
                                        }
                                    }
                                }
                            }
                        }
```

4. Experimentos realizados

Las imágenes utilizadas para valorar los resultados han sido las siguientes:

Figura 4.1: Lena: Dificultad Media

Figura 4.2: Black: Dificultad Fácil

Figura 4.3: Circle black: Dificultad Difícil

Figura 4.4: Tiger: Dificultad Difícil

5. Valoración de los resultados obtenidos

A continuación se muestran los resultados de la ejecución del algoritmo pasándole las distintas imágenes como entrada y seleccionando algunos puntos:

Figura 5.1: Lena

La imagen funciona bien si los puntos son relativamente cercanos, y como se observa sobre todo el contorno del sombrero es el que mejor se ajusta.

Figura 5.2: Black

Figura 5.3: Black 2

Si los puntos se seleccionan en las esquinas del rectángulo, o en línea recta se observa que se ajustan perfectamente. En cambio si el camino necesita hacer un giro se ajusta peor.

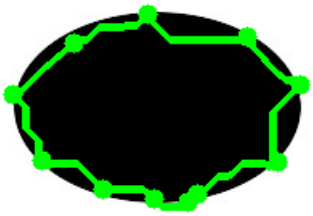

Figura 5.4: Circle black

En este caso el ajuste es muy malo, se observa como la curva nunca se ajusta y son necesarios puntos bastante cercanos para que el ajuste sea relativamente aceptable.

Figura 5.5: Tiger

Ahora el ajuste del cuerpo se observa que es aceptable, aunque en la cara el ajuste es mucho mejor.