Memoria de Prácticas

Práctica 3

Curso 2016/2017

Algoritmos Greedy

Práctica realizada por:

- Andrés Arco López
- Miguel Ángel Cantarero López
- Jorge Sánchez González
- Ismael Sánchez Torres

ÍNDICE

1.Enunciado del problema

El problema del mural:
Una imagen “grande” F se divide en N*M cuadrículas F(x,y).
Cada cuadrícula tiene el mismo tamaño y un color asignado.
Se desea realizar un mural/mosaico F’(x,y) con un total de N*M imágenes más pequeñas, cada una de ellas con un color asignado.

2. Características de un algoritmo Greedy

Las características que tienen en común los algoritmos Greedy, y que utilizaremos para dar una solución al problema son las siguientes:
· Construir la solución paso a paso, por etapas.
· En cada momento selecciona un movimiento de acuerdo con un criterio de selección (voracidad).
· No vuelve a considerar los movimientos ya seleccionados ni los modifica en posteriores etapas (miopía).

3. Estructuras de datos

En este apartado se pide diseñar e implementar las estructuras de datos necesarias para representar:

Imagen original *(Matriz de datos)* ---> Vector de enteros F(x,y) (entre 0 y 255 , según el color) de dimensión Nfilas*Mcolumnas

Vector con imágenes candidatas *(Lista de candidatos)* ---> Vector P de enteros de tamaño N*M

```
N= p.N; // Reserva de memoria nueva si es necesario y copia
M=p.M;

if (N*M>0) {

    cuadriculas_candidatas= new int[N*M];
    imagen_original = new int[N*M];

    for (unsigned int i= 0; i<N*M; i++) {

        cuadriculas_candidatas[i]= p.cuadriculas_candidatas[i];
        imagen_original[i]= p.imagen_original[i];

    }

}
```

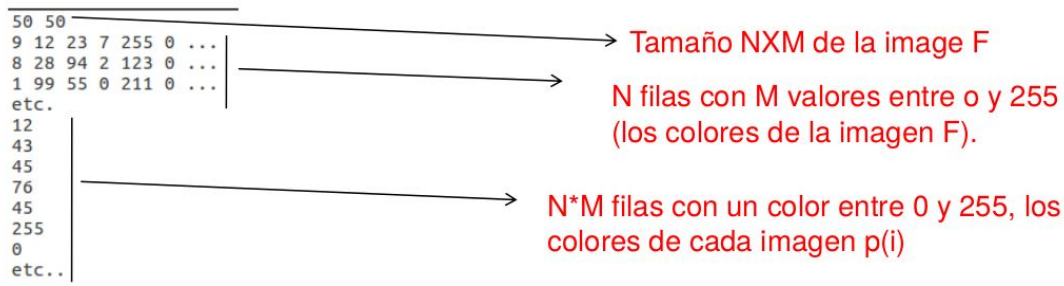
Imágenes utilizadas de las candidatas *(Lista de candidatos utilizados)* ----> Vector de booleanos de tamaño (N*M)-1

```
LC= new bool[TAM];
for (int i= 0; i<TAM; i++)
    LC[i]= false;
```

Imagen lo más parecida posible a la original utilizando las imágenes candidatas *(Matriz solución)* ---> Vector de enteros F ‘ (x,y) (entre 0 y 255 , según el color) de dimensión Nfilas*Mcolumnas, con las imágenes que correspondan del vector de candidatos

```
using namespace std;

class Problema
{
    public:
        Problema();
        Problema(const Problema & p);
        Problema & operator=(const Problema &p);
        ~Problema();

        int getValorCuadriculaCand(int i);
        int getValorCuadriculaIma(int i, int j);

        bool cargarDesdeFlujo(const char *nombreFichero); // Carga un problema
                                                          // desde el fichero dado por argumento.
                                                          // Devuelve true si ok, y false
                                                          // si error al cargarlo

        int getN() const; // Devuelve el número de filas
        int getM() const; // Devuelve el número de columnas

    protected:

        unsigned int N; // Num. de filas
        unsigned int M; // Num. de columnas

        int *cuadriculas_candidatas; // Vector con las cuadriculas candidatas
        int *imagen_original;

    private:
};
```

Adicionalmente, hemos implementado métodos para leer todos estos datos desde todos los ficheros que respeten la siguiente estructura:



```
bool Problema::cargarDesdeFlujo(const char *nombreFichero) {


    // Liberar memoria si la hubiese
    if (N*M > 0) { // Liberar la memoria previa
        delete [] cuadriculas_candidatas;
        delete [] imagen_original;
    }

    // Inicializar a problema vacío
    N= 0;
    M=0;

    ifstream fichero;

    fichero.open( nombreFichero );
    if ( !fichero )
        return false;

    fichero >> N;
    fichero >> M;
    if (N*M<=0) {
        fichero.close();
        N= 0;
        M=0;
        return false;
    }

    // Reserva de la memoria para el "N*M" nuevo
    cuadriculas_candidatas= new int[N*M];
    imagen_original= new int[N*M];

    for (unsigned int i= 0; i<N*M; i++)
        fichero >> imagen_original[i];

    for (unsigned int i= 0; i<N*M; i++)
        fichero >> cuadriculas_candidatas[i];


    fichero.close();


    return true;
}
```

Para mostrar la solución al problema, lo podemos hacer de dos formas diferentes, mostrando la salida en pantalla (como se muestra en un ejemplo del apartado 4), o enviando dicha salida a un fichero , utilizando en la terminal el operando >> fichero.dat

4. Diseño del algoritmo voraz

A la hora de plantearnos cómo podríamos hacer el diseño del algoritmo se nos ocurrieron varias formas, con las cuales probamos varios ejemplos para ver cual era más eficiente a simple vista. Contemplamos tres opciones;

-Ordenar el vector de menor a mayor y directamente colocarlo en sus respectivas casillas de la matriz empezando por la que tenga el valor más pequeño hasta llegar a colocarlas todas. Con este diseño, sin embargo, vimos que la la eficiencia era muy baja ya que con la mayoría de los datos la diferencia total entre los valores de la matriz original y la nueva era muy grande. Los otros dos tipos ofrecían siempre una diferencia menor.

-Ir cogiendo elementos del vector uno por uno en orden, e ir comparándolos con todos los elementos de la matriz original y colocándolos en la casilla en la que el valor esté más próximo al valor del elemento del vector. Este diseño a simple vista parecía más efectivo que el de ordenar el vector y así fue, la diferencia total disminuye considerablemente.

-Por último probamos a ir cogiendo uno por uno elementos de la matriz, e ir comparándolos con los del vector, escogiendo siempre el elemento del vector cuyo valor esté más próximo al valor de la casilla de la matriz. En este diseño la diferencia que nos salía con la mayoría de los datos que teníamos era la menor. Por ello este ha sido nuestro algoritmo elegido.
Sin embargo, a pesar de ser el más eficiente, a la hora de ver los resultados nos damos cuenta de que la imagen al principio sale más parecida a la original pero a medida que vamos bajando por filas, la diferencia entre la matriz original y la matriz nueva suele ir aumentando, debido a que el número de candidatos va disminuyendo, y por ello hay que ir adaptándose a ellos(aunque esto también dependerá del caso específico).

Implementación escogida:
Para la resolución del ejercicio utilizaremos las siguientes componentes Greedy (en el 3.1 se especifica la estructura de cada elemento):

Lista de candidatos: Cada uno de los elementos del vector que representa cada cuadrícula (y que tiene asociado un color).
Lista de candidatos utilizados: Los elementos del vector(cuadrículas) ya insertados en la matriz(imagen) final.
Función solución: No quedan elementos de la matriz original a comparar(a todos se les ha asignado uno del vector).
Criterio de factibilidad: El elemento candidato no se ha usado aún(no se ha usado el mismo elemento del vector dos veces).

Función de selección: El elemento candidato más cercano al de la matriz original que se está comparando.
F. objetivo: Encontrar una asignación para cada punto del mosaico F'(x,y)= i tal que sea lo más similar posible a la imagen original, es decir, minimizar:

$$\sum_{x=1}^{N} \sum_{y=1}^{M} \left| F'(x,y) - F(x,y) \right|$$

Explicación general de nuestro algoritmo:

El algoritmo es sencillo, se basa en ir cogiendo en orden de izquierda a derecha y de arriba abajo cada elemento de la matriz original; e ir comparando dicho elemento con cada uno de los elementos del vector, llevando una variable que guarde el elemento(su posición en el vector) que más cerca está del que estamos comparando hasta recorrer todo el vector. Teniendo ya esta posición, la marcamos (con otro vector) como posición utilizada(false) e introducimos su valor en la matriz solución.
Así, proseguimos con el algoritmo tomando el siguiente elemento de la matriz y comparándolo con todos los del vector salvo los marcados como utilizados(y tras compararlos introduciendo en la matriz solución el que más se acerca).

Pseudocódigo:

```
ALGORITMO Voraz(matriz de elementos de la imagen original O, y vector con cuadrículas candidatas LC)
S <- Ø
Mientras (recorremos O) hacer:
        x=LC(0)
        Mientras (recorremos LC)
                if(LC(j) más cercano a O )
                        x = LC(j)
        Fin de Mientras
        LC = LC \{x}
        O(i)=x
Fin-Mientras
Devolver S
```

Pequeño ejemplo para explicar el algoritmo:

Como podemos comprobar en este ejemplo se nos da una matriz 3x3 y correspondientemente un vector de 9 elementos(que deberemos introducir en la matriz solución).
Para aplicar el algoritmo descrito, empezamos cogiendo el elemento 0 de la matriz original y lo comparamos con todos los elementos del vector; escogemos el más cercano a 0, que en este caso es el 1, y lo introducimos en la matriz solución. Marcamos el 1 como utilizado y ahora procedemos a hacer lo mismo con el siguiente elemento de la matriz original (el 32), sin tener en cuenta ahora el 1, pues ya ha sido marcado como utilizado. Vemos que el más cercano es el 33 con lo que lo introducimos en la matriz solución y lo marcamos como utilizado. Continuamos haciéndolo hasta que no quedan más elementos en la matriz.

4.1. Eficiencia

La eficiencia teórica de nuestro algoritmo es $O(n^2)$. El número de elementos que introducimos es TAM = nº filas * nº columnas. Entonces viendo el código tenemos lo siguiente:

```
for (int i= 0; i<TAM; i++)
    LC[i]= false;
```

Este bucle for que es del orden O(TAM), es decir, O(n).

```
19      for(int i=0; i<p.getN(); i++){
20          for(int j=0; j<p.getM(); j++){
21
22
23              int valor=p.getValorCuadriculaIma(i,j);
24              int k;
25              bool lc_centinela=false;
26
27              for(k=0; k<TAM && !lc_centinela; k++){
28                  lc_centinela=(LC[k]==false);
29              }
30
31
32              int pos_buscada=k-1;
33              int diferencia=abs(p.getValorCuadriculaCand(pos_buscada)-valor);
34
35              for(; k<TAM; k++ )
36                  if(LC[k]==false)
37                      if(diferencia>abs(p.getValorCuadriculaCand(k)-valor)){
38                          diferencia=abs(p.getValorCuadriculaCand(k)-valor);
39
40                          pos_buscada=k;
41                      }
42
43              LC[pos_buscada]=true;
44              s.addCuadricula(i,j,p.getValorCuadriculaCand(pos_buscada));
45          }
46
47      }
48  //Liberamos memoria
```

Si analizamos este fragmento, podemos observar que el primer for hace ‘n’ iteraciones y el segundo for hace ‘m’ iteraciones (filas por columnas). Estos dos for se podrían cambiar por un for que haga TAM iteraciones (TAM = n*m), pero como nos hace falta los índices nos resulta más fácil trabajar así. A su vez hay dos bucles anidados de los cuales uno siempre hace TAM iteraciones y, el otro las hace en el peor de los casos. Entonces tenemos que claramente que la eficiencia de este fragmento de código es $O(TAM^2)$, es decir, $O(n^2)$. Habiendo analizado el código obtenemos que su eficiencia teórica es $O(n^2)$. (Siendo n = TAM).

$$T(n)= 2n^2+n+C$$

5. Implementación

```
#include "Algoritmos.h"
#include <cmath>
#include <iostream>

using namespace std;

Solucion AlgoritmoGreedyMural(Problema p) {
    Solucion s(p); // Solución a devolver
    bool *LC; // Lista de candidatos (false si está en LC, true si está en LCU)


    int TAM=p.getN() * p.getM(); //Tamaño del vector LC
    // Inicializar la lista de candidatos
    LC= new bool[TAM];
    for (int i= 0; i<TAM; i++)
        LC[i]= false;
    /*Algoritmo Greedy para encontrar la imagen del vector LC que más se parece a la de la matriz original
        */
    for(int i=0; i<p.getN(); i++){
        for(int j=0; j<p.getM(); j++){

//Tomamos el valor de la matriz original, inicialmente la posicion M[0][0]
            int valor=p.getValorCuadriculaIma(i,j);
            int k;

//La variable ls_centinela nos indica si el elemento LC[k] esta usado o no LC=[true,true,true,false,false]
            bool lc_centinela=false;

            for(k=0; k<TAM && !lc_centinela; k++){
                lc_centinela=(LC[k]==false);
            }


            int pos_buscada=k-1;
//Calculamos la diferencia entre nuestra imagen candidata y la original
            int diferencia=abs(p.getValorCuadriculaCand(pos_buscada)-valor);
//Ahora actualizamos la diferencia hasta que encontremos la menor entre todos los elementos de LC
            for(; k<TAM; k++ )
                if(LC[k]==false)
//Si la diferencia es menor que la que hemos calculado en iteraciones anteriores, actualizamos nuestra imagen buscada pos_buscada=k
                    if(diferencia>abs(p.getValorCuadriculaCand(k)-valor)){
                        diferencia=abs(p.getValorCuadriculaCand(k)-valor);

                        pos_buscada=k;
                    }
//La marcamos como utilizada en la LC
            LC[pos_buscada]=true;
//La añadimos a la nueva imagen
            s.addCuadricula(i,j,p.getValorCuadriculaCand(pos_buscada));
        }

    }
//Liberamos memoria

    delete [] LC;
//Return de la imagen final
    return s;
}
```