Presentación
Práctica 3

Algoritmos Greedy

Realizado por:

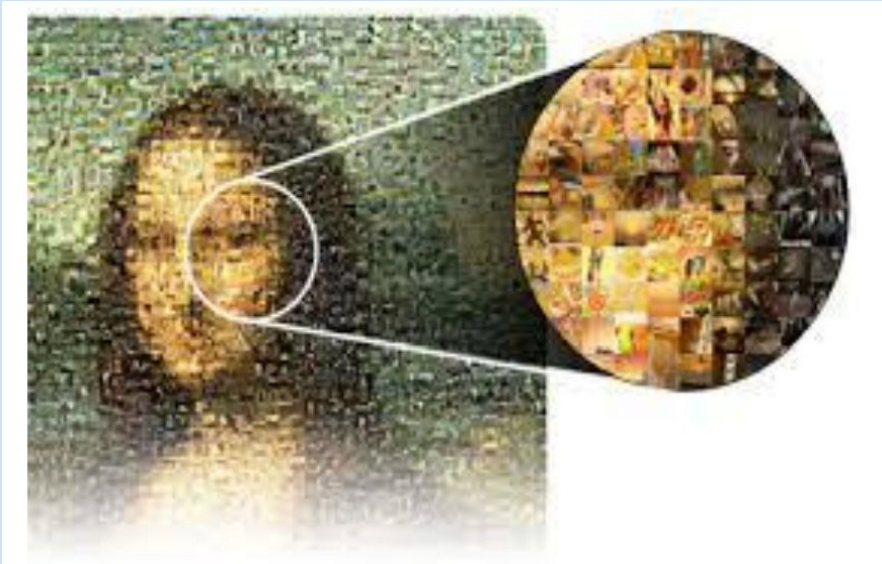
- Andrés Arco López
- Miguel Ángel Cantarero López
- Jorge Sánchez González
- Ismael Sánchez Torres

Índice

1. *Enunciado del problema*
2. *Diseño del algoritmo voraz*
3. *Estructuras de datos empleadas*
4. *Implementación*
5. *Eficiencia*

1. Enunciado del problema

Una imagen “grande” F se divide en N*M cuadrículas F(x,y).
Cada cuadrícula tiene el mismo tamaño y un color asignado(número entre 0 y 255).
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. Diseño del algoritmo voraz

A la hora de diseñar el algoritmo se nos ocurrieron **tres opciones**:

-**Ordenar el vector de la imágenes pequeñas** de menor a mayor(por el número que tienen asignado) y colocarlo en la matriz (desde las casillas de valores más pequeños hasta el más grande).
-**Coger elementos del vector uno por uno**, e ir colocándolos en las casillas de la matriz cuyo valor esté más próximo al del elemento del vector seleccionado en cada caso.
-Por último **coger los elementos de la matriz(imagen original) uno por uno** e ir comparándolos con los del vector y en cada iteración seleccionar el elemento del vector cuya diferencia es menor.

Este último fue el algoritmo elegido ya que tras varias pruebas nos dieron diferencias totales entre la matriz original y la nueva más pequeñas que usando los dos algoritmos anteriores. Además nos dimos cuenta de que era el más flexible si nos pedían resolver el problema con un vector que tuviera más cuadrículas que la imagen original.

2.1 Componentes Greedy :

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:

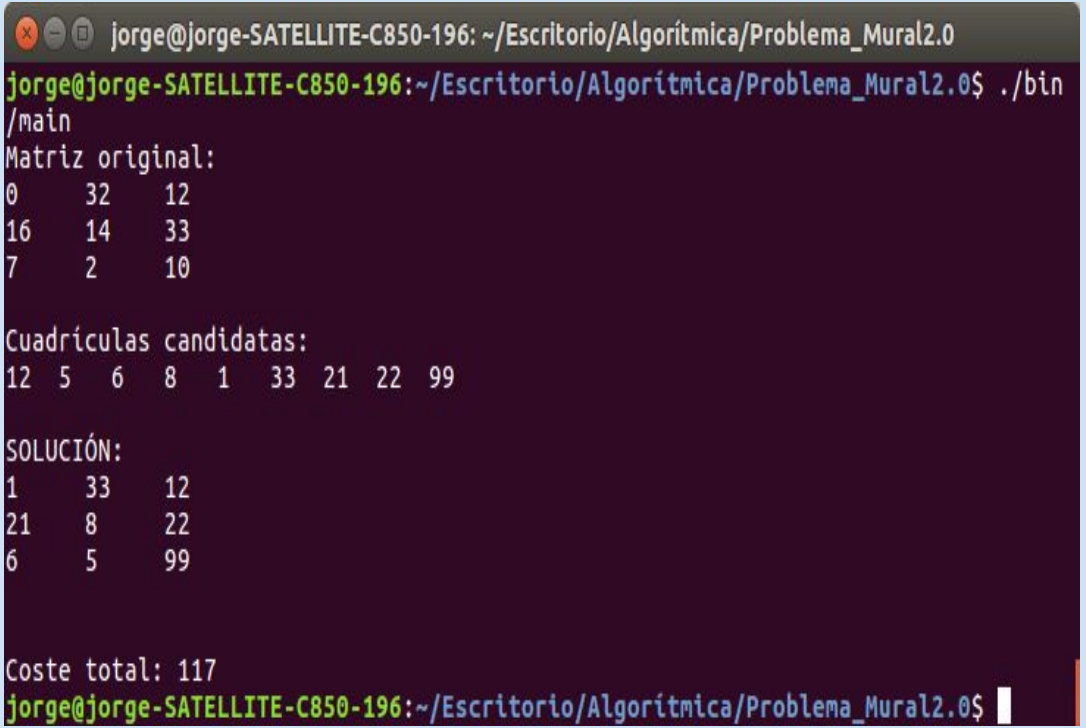
$$\sum_{x=1}^{N} \sum_{y=1}^{M} \left| F'(x,y) - F(x,y) \right|$$

2.2 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}
        S(i)=x
Fin-Mientras
Devolver S
```

2.3 Pequeño ejemplo:

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.Continuamos haciéndolo hasta que no quedan más elementos en la matriz.



3. Estructuras de datos

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

```
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:
};
```

3. Estructuras de datos

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

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

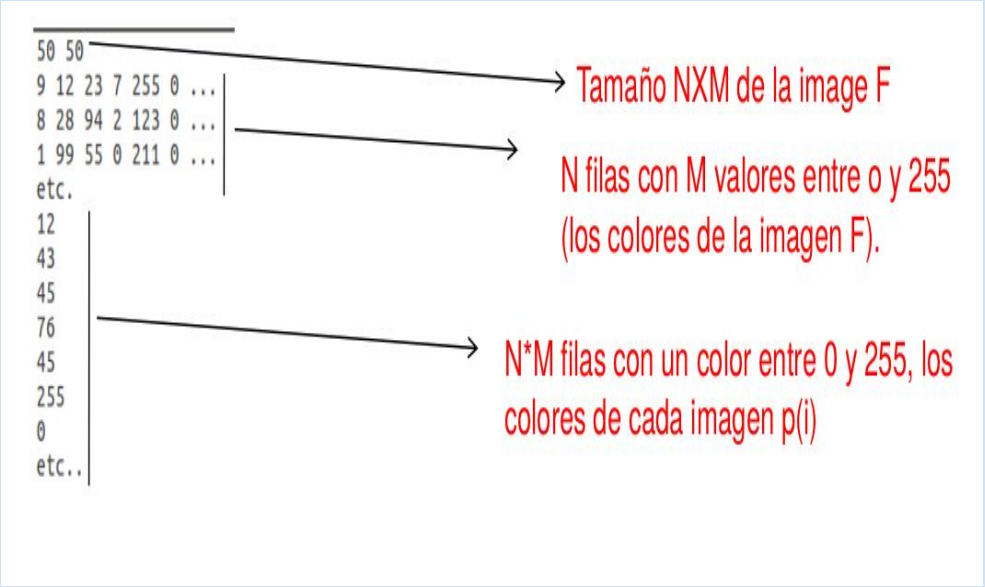
3. Estructuras de datos

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

```
Solucion::Solucion(const Problema & p) {

    N= p.getN();
    M= p.getM();

    if (N*M>0) {
        sol= new int[N*M];

        for (unsigned int i= 0; i<N*M; i++)
            sol[i]= -1;
    }
    coste=-1;
}
```

3.1 Lectura de datos

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




```
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();
```

3.2 Salida de datos

Salida estándar

```
jorge@jorge-SATELLITE-C850-196: ~/Escritorio/Algorítmica/Problema_Mural2.0
jorge@jorge-SATELLITE-C850-196:~/Escritorio/Algoritmica/Problema_Mural2.0$ ./bin
/main
Matriz original:
0     32    12
16    14    33
7     2     10

Cuadrículas candidatas:
12  5   6   8   1   33  21  22  99

SOLUCIÓN:
1     33    12
21    8     22
6     5     99


Coste total: 117
jorge@jorge-SATELLITE-C850-196:~/Escritorio/Algoritmica/Problema_Mural2.0$
```

En un fichero

```
miguel@miguel-Lenovo-Z50-70:~/Documentos/ALGORITMIA/greedy/algor
itmica$ ./bin/problemamural Problema-ejemplo.dat >> fichero.dat
```

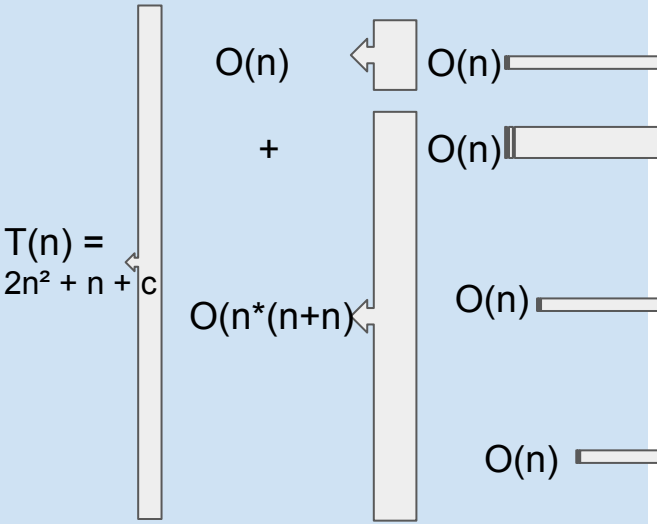
```
Matriz original:
0     32    12
16    14    33
7     2     10

Cuadriculas candidatas:
12  5   6   8   1   33  21  22  99

SOLUCIÓN:
1     33    12
21    8     22
6     5     99


Coste total: 117
```

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;
}
```

6. Eficiencia

Tomando n
como TAM=N*M



La eficiencia del algoritmo es
O(n²)

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

    for(int i=0; i<p.getN(); i++){
        for(int j=0; j<p.getM(); j++){

            int valor=p.getValorCuadriculaIma(i,j);
            int k;

            bool lc_centinela=false;

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


            int pos_buscada=k-1;
            int diferencia=abs(p.getValorCuadriculaCand(pos_buscada)-valor);
            for(; k<TAM; k++ )
                if(LC[k]==false)
                    if(diferencia>abs(p.getValorCuadriculaCand(k)-valor)){
                        diferencia=abs(p.getValorCuadriculaCand(k)-valor);

                        pos_buscada=k;
                    }
            LC[pos_buscada]=true;
            s.addCuadricula(i,j,p.getValorCuadriculaCand(pos_buscada));
        }

    }
    delete [] LC;
    return s;
}
```

FIN