

Concurso Programacion EDA  
2010-2011

Alberto Sendra Estrella  
12-05-2011

# Contents

<b>1</b>	<b>Introducción</b>	<b>3</b>
1.1	Introducción . . . . .	3
1.2	Desglose del problema . . . . .	3
<b>2</b>	<b>Lectura de Imágenes</b>	<b>3</b>
2.1	Lectura mediante ifstreams . . . . .	3
2.2	Lectura binaria . . . . .	4
<b>3</b>	<b>Obtención de Parches</b>	<b>4</b>
3.1	Metodo Normal . . . . .	4
3.2	Metodo Optimo . . . . .	5
<b>4</b>	<b>Parches Unicos</b>	<b>5</b>
4.1	Usando Ordenación . . . . .	6
4.2	Usando Tabla Hash . . . . .	6
<b>5</b>	<b>Calculo de Distancias</b>	<b>7</b>
5.1	Distancias Merge . . . . .	7
5.2	Distancias Hash . . . . .	8
5.3	Distancias de forma "Voraz" . . . . .	8
5.3.1	Optimizando aun más . . . . .	9
<b>6</b>	<b>K Menores Distancias</b>	<b>9</b>
<b>7</b>	<b>Grafo</b>	<b>9</b>
7.1	Estructura del grafo . . . . .	9
7.2	Distancia entre nodos del Grafo . . . . .	10
7.2.1	Dijkstra con Colas . . . . .	10
7.2.2	Dijkstra con Heaps . . . . .	11

# 1 Introducción

## 1.1 Introducción

Durante la realización del Concurso, el equipo usado ha sido un Macbook CoreDuo ('06), sobre el Sistema Ubuntu 10.10. El compilador usado fue Gcc, en su version 4.4.5.

Las flags de optimización utilizadas para mejorar mas el rendimiento fueron: -O3, y -fomit-frame-pointer.

El primer envio realizado al oraculo fue el dia 21-02-2011, y el primer envio válido se produjo el dia 24-02-2011, con un tiempo de 332,94. El mejor tiempo conseguido se produjo en el envio del 6 de Mayo a las 24:00 con un tiempo de 6,36s. lo que supone la 1 plaza en el concurso. Tambié me gustaria agradecer al compaero y tambien concursante Juanma, que hizo un test de prueba de 2250 imagenes que vino muy bien para mejorar las ultimas cosas en el ejercicio.

## 1.2 Desglose del problema

Para facilitar la explicación de la resolución, el ejercicio propuesto es facilmente divisible en cuatro apartados, y es esta la organización que voy a adoptar a lo largo de esta explicación.

# 2 Lectura de Imagenes

El primer problema a resolver en el Concurso es la lectura de los datos de las imagenes a procesar. A continuación veremos dos posibles metodos, siendo el segundo de ellos la version final entregada por mi mismo.

## 2.1 Lectura mediante ifstreams

Este metodo se basa en el uso de streams y en su funcionalidad de usar el operador >> para parsear los datos del stream automaticamente al formato deseado.

Lectura con fstreams

```
1 void Image::leer_ppm(std::istream &fich_in) {
2
3     const int tallafirma = 10;
4     char firma[tallafirma];
5     fich_in >> std::setw(5) >> firma;
6
7     int maxvalue, numpixels;
8     fich_in >> ancho >> alto >> maxvalue;
9     numpixels = ancho*alto;
10
11     // Crear aqui un vector con el num de patches totales.
12     int limitFilas = alto - w + 1;
13     int limitColumnas = ancho - w + 1;
14     int *datos = new Pixel[limitFilas*limitColumnas];
15
16     int r,g,b,pos=0;
17     for (int i=0; i<alto; i++) {
18         for (int j=0; j<ancho; j++) {
19             fich_in >> r >> g >> b;
20             datos[ pos++ ] = (r / c);
21             datos[ pos++ ] = (g / c);
22             datos[ pos++ ] = (b / c);
23         }
24     }
25     ....
26 }
```

Esta opción, pese a ser comoda por su sencillez y claridad, no es muy optima por varios motivos. Primero, el operador >> en un stream tiene que tener en cuenta muchas cosas y funcionar en muchos escenarios distintos por lo que necesita realizar calculos de más.

Otro inconveniente de esta solución es el hecho de estar accediendo repetidamente a un mismo fichero, esto produce pequeños micro bloqueos en la CPU durante el tiempo de acceso a disco, y esto, en ordenadores actuales supone una gran pérdida de rendimiento ya que en ese espacio de tiempo un ordenador es capaz de ejecutar muchos miles de instrucciones.

## 2.2 Lectura binaria

Este método soluciona ambos problemas expuestos en el método anterior. En primer lugar, leemos la información de los píxeles en una sola lectura, metiéndolo en un buffer.

Una vez con los datos leídos, nos encargamos de parsear esa información, que al fin y al cabo son chars en ASCII, a números enteros. Para ello me he basado en una simplificación del método de la librería estándar atoi.

### Lectura en Binario

```

1 void Image::leer_ppm(std::istream &fich_in) {
2
3     fich_in.ignore(3);
4     fich_in >> ancho >> alto;
5     fich_in.ignore(4);
6
7     int current = fich_in.tellg();
8     fich_in.seekg(0, ios::end);
9     int length = fich_in.tellg();
10    length -= current;
11    fich_in.seekg(current, ios::beg);
12
13    // allocate memory:
14    char* buffer = new char[length];
15
16    // read data as a block:
17    fich_in.read(buffer, length);
18
19    int* int_vector = new int[ancho*alto*3];
20    int num = 0, pos = 0, cont = 0;
21    bool valid = false;
22
23    while(cont < length) {
24        if (buffer[cont] == ' ' || buffer[cont] == '\n') {
25            if (valid) {
26                int_vector[pos] = (num / c);
27                valid = false;
28                num = 0;
29                pos++;
30            }
31            else {
32                num = num*10;
33                num += (int)buffer[cont];
34                num -= 48;
35                valid = true;
36            }
37            cont++;
38        }
39        delete[] buffer;
40        .....
41    }

```

## 3 Obtención de Parches

### 3.1 Metodo Normal

Es necesario aclarar que para simplificar los cálculos a lo largo del ejercicio, los parches los represento como un solo entero usando n bits para cada componente del parche y combinándolos en un solo

entero mediante los operadores  $\ll$  y el operador OR lógico.

Aunque es algo que se me había ocurrido de forma parecida desde el principio, la forma final que uso la saque de un foro, aunque creo que todos hemos acabado usando alguna variación de la misma al final.

A la hora de leer los parches, el método mas obvio y el primero que se nos viene a la cabeza es algo parecido a esto.

Caption without label

```
1  int ultimo, Sr=0, Sg=0, Sb=0;
2  int *parches = new int[totalParches];
3
4  for (int i=0; i<limitF; ++i) {
5      for (int j=0; j<limitC; ++j) {
6          Sr=Sg=Sb=0;
7          for (int k=0; k<w; ++k) {
8              pos = ancho*(i+k) + j;
9              for (int l=0; l<w; ++l) {
10                 Sr+=(datos[pos+l]);
11                 Sg+=(datos[pos+l+1]);
12                 Sb+=(datos[pos+l+2]);
13             }
14         }
15         parches[ultimo++] = (Sr << 14 | Sg << 7 | Sb);
16     }
17 }
18 delete[] parches;
```

Sin embargo, esta solución tiene dos problemas. Uno, tiene un muy mal uso de la cache, ya que estamos saltando continuamente entre filas, las cuales están en zonas alejadas de memoria, y segundo, se están repitiendo muchos cálculos ya hechos previamente para el cálculo de cada parche.

### 3.2 Metodo Optimo

Para solucionar el problema de cache, la solución es leer los datos en orden. Para ello acumulamos las  $w$  filas necesarias en un vector auxiliar de tamaño fila. A su vez, sabemos que a partir de un parche ya calculado, el siguiente parche es el resultado de restar la primera columna del parche anterior y sumar la siguiente columna, ahorrándonos así volver a calcular la acumulación de las  $w$  columnas. Para cada nueva fila debemos restar al vector de filas acumuladas auxiliar la fila:  $\text{fila} := \text{fila} - w$  y sumar la fila:  $\text{fila} := \text{fila} + 1$ .

Debido a la longitud del código para demostrar este funcionamiento, me remito a que lo mireis directamente del archivo de código `image.cc`.

Como curiosidad en este algoritmo he usado un Duff Device, estructura especialmente diseñada para la copia de buffers de memoria o similares, que combina un switch con un do while, y que como objetivo principal tiene realizar más cálculos en cada iteración del bucle, reduciendo el número de ejecuciones del bucle a costa de tener más instrucciones en él. Para una mejor explicación recomiendo leer su artículo correspondiente en la Wikipedia. Realmente apenas mejora la lectura, y en el concurso ni se nota, pero ya que lo implemente lo he dejado pese a ser menos entendible.

## 4 Parches Unicos

El siguiente problema a resolver radica en transformar el vector de parches de cada imagen en un vector de parches únicos con el número de repeticiones de cada uno. Hay muchas maneras de resolver este problema, y a lo largo del concurso debo de haber probado casi todas ellas, así que solo comentare la primera y la final.

Entre otras soluciones que no pasare a comentar detenidamente se encuentra en uso de árboles (AVL, Splays, Rojo-negros) ya que en cierto punto del concurso requería de la propiedad de obtener los parches ordenados, cosa que no me permitía una tabla Hash.

## 4.1 Usando Ordenación

Aunque mi primera idea fue una Hash, por cuestion de sencillez, la primera solución que implemente se basaba en ordenar los parches y en aprovecharme de la propiedad que hace que al estar ordenados, los parches repetidos se encuentran en posiciones contiguas del vector.

```
1 std::sort( inicioDelVector , inicioDelVector+sizeDelVector );
2 int aparicions=1;
3
4 for( int i=1; i<sizeDelVector; ++i) {
5     if ( vector[i-1] == vector[i] ) {
6         aparicions++;
7     } else {
8         vectorUnicos[ultimo++] = parche(vector[i-1], aparicions);
9         aparicions = 1;
10    }
11 }
12 if(vector[size-1] == vector[size-2]){
13     vectorUnicos[ultimo++] = parche(vector[size-1], aparicions);
14 }else{
15     vectorUnicos[ultimo++] = parche(vector[size-1], 1);
16 }
```

## 4.2 Usando Tabla Hash

Como uso la misma Hash para todas las imagenes(por separado), y con el fin de no tener que recorrer toda la hash para obtener los elementos despues de haberlos insertados todos, uso un vector de nodos del maximo numero de parches posibles ( es decir, que no se repitiese ningun parche ).

Al insertar un nodo nuevo en la tabla hash, en vez de crearlo con new lo que hago es apuntar a la siguiente posicion libre en este vector de nodos. De esta manera, cuando todos los elementos han sido insertados , ya los tengo todos en un vector, el cual puedo pasar al siguiente metodo que lo necesite. Por lo demas, el funcionamiento de esta tabla Hash es igual al que hemos visto en clase por lo que solo voy a pasar a comentar el metodo insertar y el de reset( que me permite reusar la misma hash para todas las imagenes).

```
1 #define hash32(x) ((x)*2654435761u)
2 #define H_BITS 21 // Hashtable size
3 #define H_SHIFT (32-H_BITS)
4
5 void hash_table_2::insertarVal(unsigned int hash_value){
6     unsigned int p = (unsigned int) hash32(hash_value) >> H_SHIFT;
7
8     if ( !table[p] || !search(hash_value, p) ) {
9         // La imagen NO esta en la tabla
10        hash_node_2 *aux = table[p];
11        // freeNode apunta a la primera posicion libre
12        // del vector de nodos
13        table[p] = (hash_node_2*) freeNode;
14        table[p]->hash_value = hash_value;
15        table[p]->apariciones = 1;
16        table[p]->next = aux;
17        // Apuntamos a la sig. posicion libre
18        freeNode++;
19        numElem++;
20    }
21
22 }
```

```

24 void reset() {
25     freeNode = (hash_node_2*)listOfNodes;
26     memset(table, 0, sizeof(hash_node_2)*tableSize);
27     numElem=0;
28 }

```

La funcion de hash que utilizo aqui es una muy simple y a la vez eficaz, la multiplicativa de Knuth. Al hash value obtenido le realizo un desplazamiento a derechas para quedarme con N bits superiores. Siendo N el numero de bits de tamao que tiene la tabla Hash. Tamao =  $(1 < N \text{ Bits})$ . La funcion reset se encarga de resetear el puntero al principio del vector de nodos, poner todos los punteros de la tabla a cero, y resetear el contador del numero de Elementos insertados a cero.

## 5 Calculo de Distancias

EL calculo de las distancias es la parte más costosa del ejercicio sin duda, y la manera en la que se resuelva afectara radicalmente al tiempo final. Voy a comentar tres maneras totalmente distintas de resolverlo, de la primera a la última hay una diferencia de mas de 100 seg. en el oráculo.

### 5.1 Distancias Merge

La primera de las soluciones que voy a plantear este problema se basa en una adaptacion del algoritmo merge del mergesort, y tiene como requerimiento que los vectores de parches únicos de las imagenes esten ordenados, por lo que hay que ordenarlos de ante mano si aun no los tenemos ordenados, aumentando as el coste aún más.

Otro dato a destacar es que solo es necesario realizar el calculo de la distancia una vez entre dos imagenes (  $\text{distancia}(\text{img1}, \text{img2}) == \text{distancia}(\text{img2}, \text{img1})$  ), por lo que solamente es necesario recorrer la triangular superior de la matriz.

Caption without label

```

1  /*Algoritmo basado en el Merge del Mergesort*/
2  /*Recorre dos vectores ordenados crecientemente, y va aumentando el valor ↵
   distancia*/
3  /*Si son iguales -> la diferencia de sus apariciones en valor absoluto*/
4  /*Si son distintos, suma su num de apariciones*/
5  int calcularDst(Patch *A, int tamA, Patch *B, int tamB){
6      int distancia = 0;
7      int posA = 0, posB = 0;
8      while ((posA < tamA) && (posB < tamB)){
9          if(A[posA] < B[posB]){
10             distancia+=A[posA].apariciones;
11             posA++;
12          }else if(B[posB] < A[posA]){
13             distancia+=B[posB].apariciones;
14             posB++;
15          }else{
16             distancia+=(int)(abs((A[posA].apariciones) - (B[posB].↵
               apariciones)));
17             posB++;
18             posA++;
19          }
20      }
21      while (posA< tamA){
22          distancia+=A[posA].apariciones;
23          posA++;
24      }
25      while (posB< tamB){
26          distancia+=B[posB].apariciones;
27          posB++;
28      }
29      return distancia;
30 }

```

El algoritmo recorre en paralelo ambos vectores de parches, y va incrementando el valor de la variable distancia.

## 5.2 Distancias Hash

Otra manera de resolver el mismo problema seria haber creado previamente una tabla hash por cada imagen a la hora de averiguar los parches únicos, y ahora solamente habria que comprobar si los parches de una imagen estan en la hash de la otra, de ser asi se sumaria a la distancia la diferencia en valor absoluto, y si no, el numero de apariciones de ese parche.

## 5.3 Distancias de forma "Voraz"

Esta manera me costo mucho tiempo averiguarla, pero realmente valió la pena porque supuso un salto enorme.

La idea de este algoritmo es inicializar la distancias a un máximo teorico que nunca se va a sobrepasar, y conforme vayamos metiendo los parches, ir relajando estas distancias de manera que al haber insertado todos los parches una sola vez (coste lineal), las distancias esten ya calculadas.

Para ello inicializamos la matriz de Distancias con su máxima distancia posible, que es la suma de los parches que hay en ambas imagenes, ya que la máxima distancia se da cuando no coincide ningún parche. La estructura utilizada para ir metiendo los parches es una otra tabla hash, en la que cada nodo contiene el parche, y un lista con todos los indices de imagenes en las que aparece ese parche y su numero de apariciones. De este modo, cuando metemos un parche nuevo, si ya esta en la tabla, lo que hay que hacer es actualizar las distancias solamente de esas imagenes en las que tambien aparece ese parche.

### Hash de Distancias

```
1 void hash_table::insertUniq(unsigned int hash_value, int apar) {
2     if ( !table[hash_value] ) {
3         // La imagen NO esta en la tabla
4         index_node auxil;
5         auxil.index = current;
6         auxil.apariciones = apar;
7         table[hash_value] = new std::vector<index_node>;
8         table[hash_value]->reserve(16);
9         table[hash_value]->push_back(auxil);
10        numElem++;
11    } else {
12        average(hash_value, apar);
13    }
14 }
15 void hash_table::average (unsigned int hash_value, int apar) {
16     for (std::vector<index_node>::iterator it = table[hash_value]->begin(); it
17         != table[hash_value]->end(); ++it) {
18         // Calculamos la diferencia de apariciones en valor absoluto
19         // == abs(a,b) pero mas eficiente
20         int dif = apar - (*it).apariciones;
21         int const mask = dif >> 31;
22         dif = (dif + mask) ^ mask;
23         // restamos el numero de apariciones en ambas
24         // y sumamos la dif en valor absoluto
25         dif = dif - apar - (*it).apariciones;
26         aux[ (*it).index ] += dif;
27     }
28     index_node auxil;
29     auxil.index = current;
30     auxil.apariciones = apar;
31     table[hash_value]->push_back(auxil);
32 }
```



### 5.3.1 Optimizando aun más

Este método me hizo bajar de 55 s ( calculando las distancias con una hash por imagen) a 16s, que no estaba mal, sin embargo había una optimización muy facil de realizar. Consiste en cambiar la lista de indices y apariciones, por un `std::vector`, de maera que todos los datos se encuentran contiguos en memoria y aprovechamos mejor la localidad de datos y el uso de la cache del ordenador. Este pequeno cambio me hizo bajar hasta los 10s.

Otra optimización más es posible, sin embargo no aporta mucho ( 0,7s ), y además hace que ciertos casos no funcionen. Se trata de aprovechar el hecho de que a lo alrgo del concurso, conseguimos averiguar los datos usados por el oraculo, los cuales eran 3 5 64. Esto nos permite observar que el mximo valor posible para cada componente del parche no puede ser superior a 75,

$(255/c) * (w*w) = 75$ , de manera que el parche lo podemos representar únicamente con 7 bits por componente, necesitando asi 21 bits para el parche, lo que nos permite hacer una tabla hash de tamaño  $1 < 21$ , y por lo tanto, permitiendonos tener una hash sin colisiones.

## 6 K Menores Distancias

El cálculo de las K menores Distancias que seran insertadas en el grafo es algo bastante trivial, y cualquiera de los metodos empleados deberia de ser lo suficientemente bueno como para no notar una diferencia en el Concurso más alla de un par de decimas. yo personalmente he probado 3 formas, las 3 usando metodos de la libreria STL, por lo que son sencillas y rápidas de probar e implementar

```
1  for ( i=0;i<numeroImagenes;++i) {
2      //Metodo 1
3      std::sort ( matrixDistancias [ i ] , matrixDistancias [ i ]+numeroImagenes );
4      //Metodo 2
5      std::partial_sort ( matrixDistancias [ i ] , matrixDistancias [ i ]+(k+1) ,↵
6                          matrixDistancias [ i ]+numeroImagenes );
7
8      for ( j=1;j<=k ;++j ) {
9          //Metodo 3
10         std::nth_element ( matrixDistancias [ i ] , matrixDistancias [ i ]+j , ↵
11                             matrixDistancias [ i ] + num );
12         Dijs.insertar ( i , matrixDistancias [ i ] [ j ].img , matrixDistancias [ i ] [ j ].↵
13                         distance );
14     }
15 }
```

El método 1 simplemente ordena todas las distancias, por lo que es el menos óptimo de los 3.

El método 2 es el ms óptimo, realiza una ordenación parcial de solamente los k elemntos necesarios en coste n. El méotdo 3 teoricamente tiene el mismo coste n, aunque se tiene que llamar k veces, una por cada elemento a insertar. Se basa en el algoritmo de Order Statics, bastante parecido al select del quicksort, y lo que hace es poner en la posición n el n elemento menor indicado.

## 7 Grafo

### 7.1 Estructura del grafo

Para representar el grafo, uso una pequeña variancion de la representacion mediante listas de adyacencia, ya que en vez de usar una lista uso un `std::vector`, lo que mejora un pelin el recorrido de las listas.

A la hora de insertar las aristas en el grafo, como sabemos que el grafo es unidireccional, todas las aristas las tenemos que insertar 2 veces, una por cada posible dirección de ese camino o arista.

## Inserción de Aristas

```
1 void grafo::insertar(int u, int v, int w){
2
3     g_node aux;
4
5     aux.peso = w;
6     aux.dest = v;
7     vec[u]->push_back(aux);
8
9     aux.dest = u;
10    vec[v]->push_back(aux);
11 }
```

## 7.2 Distancia entre nodos del Grafo

La ultima parte que se requiere del ejercicio del Concurso es un problema muy comun en teoria de grafos, conocido en ingles como "All-pairs Shortest Path" y es el cálculo de la distancia de cada vertice, con todos los otros vertices.

Hay varias maneras de resolver este problema, como por ejemplo usando los algoritmos de Floyd-Warshall o de Johnson, sin embargo no hablare de estos métodos ya que no los he probado.

El método que he usado en el concurso ha sido el de usar el algoritmo de Dijkstra, que resuelve las distancias entre un nodo dado y todos los demas, y llamando una vez por cada nodo del grafo, e imprimiendo en cada iteración por pantalla las distancias.

En mi clase grafo podreis encontrar 3 versiones del algoritmo, la primera de ellas es la que se da en clase por lo que no la comentare.

### 7.2.1 Dijkstra con Colas

Este algoritmo me lo encuentre en internet, y para mi sorpresa, es el que va más rápido para todas las pruebas que he realizado, aunque en el concurso apenas se diferencia de el siguiente que explicare a continuacin.

La verdad es que aun no soy capaz de explicar como va incluso más rápido que la version con heaps doblemente enlazados.

## Dijkstra v1

```
1 void grafo::dijkstra2() {
2
3     short queue[tam];
4     char inQueue[tam];
5
6     for(int s=0; s < tam; ++s) {
7
8         int begq = 0, endq = 0;
9         int i, mini;
10
11         for (i = 0; i < tam; ++i) {
12             dist[i] = INFIN;
13             inQueue[i] = 0;
14         }
15
16         dist[s] = 0;
17         queue[endq] = s;
18         endq++;
19
20         while (begq != endq) {
21
22             mini = queue[begq];
23
24             begq = (begq + 1) % tam;
25             inQueue[mini] = 0;
```

```

28         for( std::vector<g_node>::iterator it = vec[mini]->begin() ; it !=
29             != vec[mini]->end(); it++) {
30             if (dist[mini] + (*it).peso < dist[(*)it).dest]) {
31                 dist[ (*it).dest ] = dist[ mini ] + (*it).peso;
32                 if (!inQueue[ (*it).dest ]) {
33                     queue[ endq ] = (*it).dest;
34                     endq = (endq + 1) % tam;
35                     inQueue[ (*it).dest ] = 1;
36                 }
37             }
38         }
39         imprimirDistancias();
40     }
41 }

```

### 7.2.2 Dijkstra con Heaps

Esta version esta basada en la implemetacin de Dijkstra optimizada mediante MinHeaps o Priority Queues. El MinHeap en este caso es un heap como los estudiados en clase pero que con vector auxiliar que nos da la posicion de cada elemento en el vector con coste uno, para as optimizar la funcion decrease-key.

#### Dijkstra v2

```

1 void grafo::dijkstra5 () {
2     Pqueue heap;
3     heap.setsize( tam );
4     estado color[tam];
5
6     for( int s=0; s < tam; ++s) {
7         for (int i = 0; i < tam; i++){
8             dist[i] = INFIN;
9             color[i] = WHITE;
10        }
11        color[s] = GRAY;
12        dist[s] = 0;
13        heap.insert(0,s);
14
15        while(!heap.empty() ) {
16            int key;
17            int node;
18            heap.extract(key,node);
19
20            for( std::vector<g_node>::iterator it = vec[node]->begin() ; it !=
21                != vec[node]->end(); it++) {
22                if( (*it).peso + dist[node] < dist[(*)it).dest] ){
23                    dist[(*)it).dest] = (*it).peso + dist[node];
24                    if( color[(*)it).dest] == WHITE){
25                        color[(*)it).dest] = GRAY;
26                        heap.insert( dist[(*)it).dest], (*it).dest);
27                    } else if( color[(*)it).dest] == GRAY){
28                        heap.decreasekey(dist[(*)it).dest], (*it).dest);
29                    }
30                }
31            }
32            color[node] = BLACK;
33        }
34        imprimirDistancias();
35        heap.clear();
36    }
}

```