



CS3102  
ESTRUCTURAS DE DATOS AVANZADAS

**Agrupamiento de Rostros por Similitud**  
Aplicación de Fibonacci Heap

Prof. Cristian Lopez del Alamo

---

Jorge Mayna  
Roosevelt Ubaldo  
Enrique Sobrados

25 de octubre de 2020

# 1. Introducción

Imaginemos que queremos validar la identidad de una persona, y que para hacerlo cogemos su documento de identidad y verificamos que su rostro corresponda con la foto de su documento de identidad. Ahora bien, imaginemos que queremos realizar la misma tarea para los más 30 millones de peruanos. La tarea además de no ser nada eficiente, sería hasta imposible sin la ayuda de los ordenadores. Es ahí donde la búsqueda de algoritmos y estructuras de datos eficientes para realizar esa tarea empiezan a tomar una vital importancia, ya que en nuestro día a día la detección de rostros ya no es más una ciencia ficción, es parte de nuestro presente.

Algunas aplicaciones de estas tecnologías son la validación de la identidad de las personas al momento de realizar trámites documentarios, acceder a dispositivos personales, controlar los aforos en locales, verificar el cumplimiento de normas (detección de mascarillas), identificar el estado de ánimo, ayudar a la detección de enfermedades, etc. En el presente informe se mostrará la aplicación de la estructura de datos Fibonacci Heap y el algoritmo de Kruskal para el agrupamiento por similitud de más de tres mil rostros. Para ello se empleó la función Haar Wavelet para extraer las características y se aplicó diferentes métodos para calcular las distancias entre estos vectores.

# 2. Implementación

El código fuente de la implementación realizada se encuentra en el siguiente enlace.

## 2.1. Extracción de Vectores Característicos

La extracción de características la haremos empleando la librería CImg. Para ello unificaremos los tamaños de todas nuestras imágenes de prueba a  $128 \times 128$  píxeles, y luego le aplicaremos la función Haar Wavelet con un corte de 2, 3 y 4 respectivamente. Finalmente, recortaremos todas las imágenes a matrices de  $16 \times 16$  y retornaremos un vector de flotantes del promedio de valores de los tres canales.

A partir de los vectores característicos de cada una de las imágenes crearemos un grafo densamente conectado de las distancias de todos contra todos los vectores característicos de las imágenes del dataset. Para ello emplearemos una estructura grafo para almacenar dicha información.

## 2.2. Estructura Grafo

La clase graph se implementó usando un vector de Nodos. Cada nodo contiene el nombre de la imagen que representa, su vector característico y las aristas que salen de dicho nodo.

```
class Graph{
private:
    vector<Nodo*> nodos;
public:
    void insert( string filename ){
        Nodo* newNode = new Nodo( filename );
        nodos.push_back( newNode );
    }

    vector<Nodo*> getNodos() {
        return nodos;
    }

    void createEdges( int dist_type ){
```

```

map <Nodo*, bool> visited;
for(size_t i=0; i <nodos.size() ; i++){
    for(size_t j=0; j < nodos.size(); j++){
        if(i == j || visited[nodos[j]]) continue;
        auto newEdge = new Arista(nodos[i], nodos[j], dist_type);
        nodos[i] ->push_edge(newEdge);
    }
    visited[nodos[i]] = true;
}
};


```

Los métodos más importantes para esta clase serán:

- **Insert**

A esta función se le pasa el nombre de una imagen como parámetro, se halla su vector característico y finalmente se inserta al grafo en forma de Nodo.

- **createEdges**

Esta función se encarga de crear todas las aristas del grafo. Es así como con  $n$  imágenes tendremos  $\binom{n}{2} = \frac{n(n-1)}{2}$  aristas.

Dado que las imágenes que se parezcan más entre sí tendrán una distancia menor entre sus vectores característicos, emplearemos un Fibonacci Heap para garantizar la extracción de la distancia mínima en  $O(\log m)$ , siendo  $m$  el número de aristas.

```

vector<float> Vectorizar(string filename, int width, int height, int cuts=3)
{
    vector<float> R;
    CImg<float> img(filename.c_str());
    img.resize(width, height);
    CImg<float> img2 = img.haar(false, cuts);
    CImg<float> img3 = img2.crop(0, 0, 16, 16);
    cimg_forXY(img3, x, y)
    {
        R.push_back((img(x, y, 0) + img(x, y, 1) + img(x, y, 2)) / 3);
    }
    return R;
}

```

### 2.3. Fibonacci Heap

Construiremos esta estructura a partir de cada uno de los elementos de nuestro grafo de aristas. La ventaja de esta estructura es que aguarda la operación de compactar para cuando se extrae el elemento mínimo, en lugar de ejecutarla cada vez que se realiza una inserción.

Las operaciones que más emplearemos de esta estructura serán **Insertion** y **ExtractMin**, cuyas complejidades son  $O(1)$  y  $O(\log m)$  respectivamente.

```

void ExtractMin()
{
    for(auto it : (*minElem)->m_Hijos){

```

```

    it->m_pPadre = nullptr;
    m_heap.push_back(it);
}

m_heap.remove(*minElem);
// delete *minElem;
this->m_size--;
}

Compactar();
min = (double)MAXDOUBLE;
for(auto it = m_heap.begin(); it != m_heap.end() ; it++){
    if( (*it)->m_key->get_value() < min){
        this->minElem = it;
        min = (*minElem)->m_key->get_value();
    }
}
}

```

Dado que las imágenes más similares entre sí tenderán a tener vectores característicos semejantes, si construimos un árbol de mínima expansión notaremos pequeños cúmulos entre las imágenes que comparten características similares. Para ello emplearemos el algoritmo de Kruskal para construir dicho árbol.

## 2.4. Árbol de Mínima Expansión (Kruskal)

Este algoritmo empleará la estructura Fibonacci Heap para extraer las aristas mínimas y un Disjoin Set para verificar la no existencia de ciclos en la creación del árbol de mínima expansión.

La estructura DisjointSet tiene 3 funciones:

- **makeSet:**  
A esta función se le pasa el vector de nodos del grafo y se encarga de crear un set por cada nodo
- **Find:**  $O(n)$   
Desde un nodo se va subiendo padre en padre hasta llegar al root el cual se retorna.
- **Union:**  $O(1)$   
Se encarga de juntar 2 sets diferentes en uno solo.

La implementación de Kruskal con disjoint sets es la siguiente:

```

template <typename T>
vector<Arista*> kruskal(FibonacciHeap<T>& FB, Graph& graph)
{
    DisjointSet disset;
    disset.makeSet(graph.getNodos());
    int nodos = graph.size()-1;
    vector <Arista *> aristas;

```

```

int cont = 0;
while(nodos){
    auto min = FB.Get_Min();
    FB.ExtractMin();
    auto dsTo = disset.Find(min->getTo());
    auto dsFrom = disset.Find(min->getFrom());
    if(dsTo == dsFrom){
        continue;
    }
    else{
        cont++;
        aristas.push_back(min);
        nodos--;
        disset.Union(dsTo,dsFrom);
    }
}
graph.setEdges(aristas);
return aristas;
}

```

Este algoritmo ejecutará  $m$  veces un llamado a la función **ExtractMin** en  $O(\log m)$  y a la función **Find** en  $O(\log n)$  del disjoint set. Por lo tanto la complejidad final de este algoritmo será:

$$\begin{aligned}
T(n) &= O(m \cdot (O(\log m) + 2 \cdot O(\log n))) \\
&= O(n^2 \cdot (O(\log n^2) + O(\log n))) \\
&= O(n^2 \cdot (O(\log n))) \\
T(n) &= O(n^2 \cdot \log n)
\end{aligned} \tag{1}$$

### 3. Experimentación

Se realizaron 3 diferentes experimentos en los cuales se utilizaron una muestra de 420 imágenes, que corresponden a 21 personas diferentes. Para cada experimento se midieron los tiempos de ejecución para cada una de las 3 formas de calcular la distancia entre vectores característicos (Manhattan, Euclidean y Minkowski). Por cada modalidad de distancia se tomaron 3 muestras para tener un promedio de el tiempo de ejecución.

Los resultados de los experimentos a continuación:

Distancia	Experimento 1 (s)	Experimento 2 (s)	Experimento 3 (s)	Promedio (s)
Manhattan	1.78599	2.00267	2.01452	1.9343
Euclidean	2.04901	1.75954	1.76937	1.8593
Minkowski	3.35699	3.26332	3.30637	3.3088

Cuadro 1: Vector característico con función Haar Wavelet de dos cortes

Distancia	Experimento 1 (s)	Experimento 2 (s)	Experimento 3 (s)	Promedio (s)
Manhattan	1.76129	1.72684	1.72599	1.73804
Euclidean	1.91503	1.84638	1.86282	1.87474
Minkowski	3.2578	3.26093	3.22726	3.24866

Cuadro 2: Vector característico con función Haar Wavelet de tres cortes

Distancia	Experimento 1 (s)	Experimento 2 (s)	Experimento 3 (s)	Promedio (s)
Manhattan	1.77496	1.80097	1.78447	1.7868
Euclidean	2.0418	2.01011	2.09684	2.0496
Minkowski	3.40003	3.35372	3.55609	3.4366

Cuadro 3: Vector característico con función Haar Wavelet de cuatro cortes

Los resultados del grafo con 3 imágenes se encuentran en los anexos.

- Anexo 1: Resultado Euclidean.
- Anexo 2: Resultado Manhattan.
- Anexo 3: Resultado Minkowski.

Luego de analizar los resultados los cuales se encuentran en la carpeta evince en el repositorio. Se puede observar que los resultados con 2 cortes no son buenos para ninguna distancia, ya que, al momento de quitar las aristas más largas se quitan algunas que no son entre diferentes imágenes. Por otro lado, con 3 y 4 imágenes los clústeres se forman bien y es fácil observar que las imágenes son las mismas. En adición, podemos ver que el promedio de tiempo de ejecución va en aumento con respecto a los cortes y los resultados siendo el mejor en tiempo de ejecución con 2 cortes y el peor con 4 cortes. Por lo tanto, con 3 cortes se obtiene el mejor resultado ya que el promedio de tiempo de ejecución es bueno y el resultado es el adecuado.

## 4. Conclusiones

Tras haber realizado distintas pruebas usando 3 tipos diferentes de distancias. Se observó que los resultados obtenidos eran muy similares entre ellos. Esto debido posiblemente a que los vectores característicos de las imágenes de un mismo grupo no estaban a mucha distancia entre ellos, haciendo casi imperceptible las diferencias al calcular las distancias con diferentes métodos.

## Referencias

- [1] wikipedia.org: Kruskal's Algorithm, [https://en.wikipedia.org/wiki/Kruskal's\\_algorithm](https://en.wikipedia.org/wiki/Kruskal's_algorithm)
- [2] Tutorialspoint.com:, Disjointsets. <https://www.tutorialspoint.com/cplusplus-program-to-implement-disjoint-set-data-structure>

## 5. Anexos

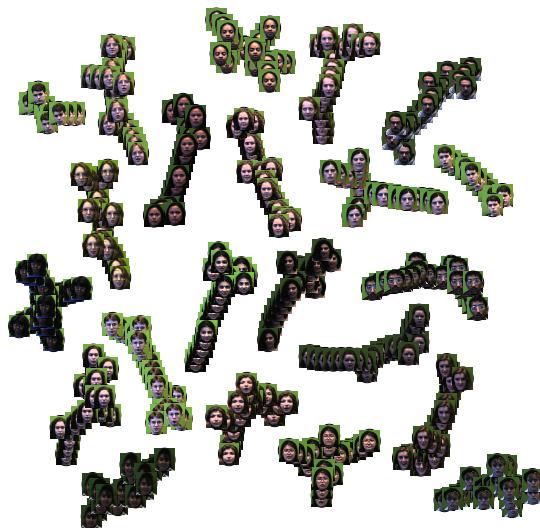


Figura 1: Distancia Manhattan con 3 cortes

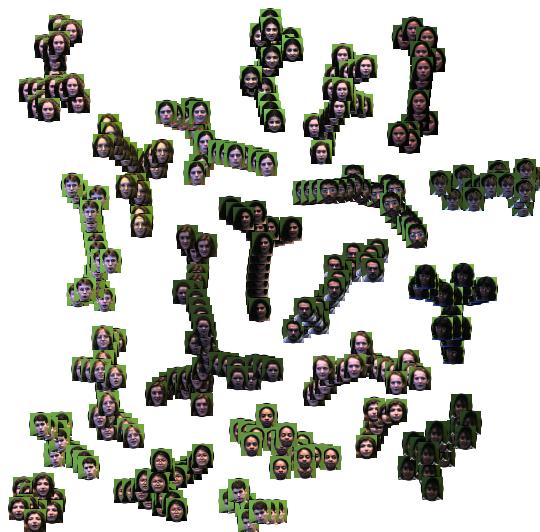


Figura 2: Distancia Minkowski con 3 cortes

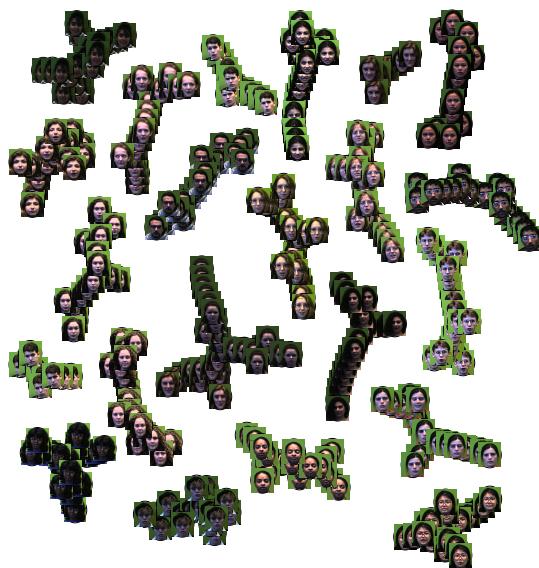


Figura 3: Distancia Euclidean con 3 cortes