Fibonacci Heap

Indhira Ramirez, Gabriel Spranger y Mateo Noel

October 25, 2020

Link al repo: `https://github.com/indhira15/FibonacciHeap`

1 Objetivos

- Usar un Fibonacci Heap para agilizar la generación del MST con el algoritmo de Kruskall, debido a la complejidad constante del ExtractMin del Fibonacci Heap.
- Generar un archivo pdf para visualizar el MST sin sus k aristas más grandes.
- Variar el número de cortes en el Haar Wavelent, y experimentar que ocurre si tomas escalas más pequeñas o si combinar 4 escalas en una sola.
- Probar la generación del grafo con distintos tipos de distancias.

2 Estructuras de datos

2.1 Fibonacci Heap

```
#define NodoT NodoF<T,D>
template<class T, class D>
class NodoF{
public:
    int rank;
    T key;
    list<NodoT*> children;
    NodoT *padre;
    D *data;
    NodoF(T value)
    NodoF(T value, D *dat)
    D* getData()
};
```

Cada nodo tiene un dato asociado a un key. También tiene un rank, su lista de hijos y su puntero a padre. Cuando creamos un nodo y lo insertamos en el heap, es ahora considerado como el root de un árbol que puede tener desde rank 0, cuando es el único nodo en su arbol, hasta nivel log2(n).

```
template<class T, class D>
class Fibonacci_heap{
private:
    int fh_size;
    list<NodoT*> fb_heap;
    NodoT * nodo_min;

public:
    Fibonacci_heap()
    int get_size()
    NodoT* Insert(T key, D *data)
    NodoT* Insert(T key)
    NodoT* GetNewMinNodo()
    NodoT* GetMinNodo()
    NodoT* DeleteMin()
    NodoT* Unir_dos_nodos(NodoT* &a, NodoT* &b)
    void Compactar()
};
```

El Fibonacci heap tiene el numero de nodos que se han insertado en el heap, la lista de heap, y un puntero al nodo con menor key. Los nodos en la lista del heap siempre son considerados nodos raíces y los hijos de estos son solo nodos.

2.2 Arista

Las aristas tienen como nodos vectores característicos, lo que caracteriza cada imagen, y la distancia que hay entre estos dos la cual esta dada por que tan parecidos son estos vectores característicos, mientras más se parezcan, menor será la distancia.

```
template <typename A>
class Arista {
public:
     VectorCaracteristico<A>* nodo1;
     VectorCaracteristico<A>* nodo2;
     float weight;

     Arista(VectorCaracteristico<A>* nodo1,
        VectorCaracteristico<A>* nodo2, float weight)
};
```

2.3 Vector característico

```
template <typename T>
class VectorCaracteristico {
      vector<T>* vc;
public:
      char imgPath[100];
      VectorCaracteristico(const char* imgPath) {
            strcpy(this->imgPath, imgPath);
            CImg<T> A(imgPath);
            CImg<T> B = A.haar(false, 1);
            CImg<T> C = B.crop(0, 0, 89, 99);
            this->vc = vectorizar(C);
      }
      void printName()
      vector<T>* get()
private:
      vector<T>* vectorizar(CImg<T>& img)
};
```

El objeto VectorCaracteristico guarda el vector característico de cada imagen y el path para saber la dirección de la imagen.

3 Implementación de Fibonacci heap

En esta sección describiremos la implementación de las funciones del Fibonacci heap y sus respectivas complejidades.

3.1 Get tamaño

Get_size() retorna el número de nodos que tiene todo el Fibonacci heap, contando tanto nodos raíces como nodos comunes. Su complejidad es O(1).

3.2 Get mínimo nodo

GetMinNodo() retorna el puntero del nodo con menor key, que ya esta guardado en la estructura como nodo_min. Su complejidad es O(1), ya que siempre tenemos un puntero apuntando al nodo mínimo.

3.3 Insert

El insert crea un nodo con el key y data dados. Después lo agrega al heap haciendo push_back. Verificamos si este tiene key menor al nodo minimo, si es así, actualizamos el nodo minimo. Su complejidad es O(1), porque solo hacemos un push_back a una lista.

3.4 Eliminar nodo mínimo

DeleteMin() borra el nodo minimo del heap. Para eliminar el nodo mínimo primero hay que insertar en el heap todos sus hijos, removemos el nodo mínimo del heap, compactamos, y buscamos un nuevo mínimo, ya que siempre tenemos que tener el puntero nodo_min actualizado. Su complejidad se debe a la función Compactar() y es O(log(n)) amortizado, se explicara posteriormente en la función compactar.

3.5 Get nuevo nodo mínimo

GetNewMinNodo() nos devuelve un puntero al mínimo nodo después de iterar por todo el heap. Lo usamos después de eliminar el nodo mínimo ya que siempre hay que tener el puntero al nodo mínimo actualizado. Al eliminar el nodo mínimo, el máximo tamaño de la lista heap es log2(n). La complejidad de esta función es O(log2(n)) porque iteramos en esta.

3.6 Compactar

La función compactar() recorre la lista nodos del Fibonacci heap, el cual esta compuesto por raíces de arboles. Este verificar si existen nodos raíces con el mismo rank, si es así, los une, aumentando el rank en 1. hasta el momento no existía un nodo de igual rank, pero ahora puede existir, por ello hay que volver a verificar este nodo de nuevo.
La complejidad del compactar dependerá de cuantos nodos raíz tiene. El peor de los casos para la función compactar se da cuando el numero de nodos totales sea igual al numero de nodos raíces. En este caso el compactar nos llevaría O(n), ya que tenemos que recorrer toda la lista para comparar todos los rank. Pero, podemos observar que este caso solo se da para el primer Delete_min(), donde se hace hace compactar por primera vez después de haber insertado solo haciendo push_back. Luego de hacer el compactar tendremos como máximo log2(n) nodos raíces y las próximas llamadas de esta función será mucho mas rápida con O(log(n)) como complejidad amortizada por esto.

3.7 Unir dos nodos

Esta función une dos nodos dependiendo de su key, el nodo con el menor key se vuelve padre del de mayor key. Su tiempo de ejecución es O(1).

4 Implementación de vector característico

Para obtener el vector característico de cada imagen, iteramos sobre cada pixel de la imagen, y por cada píxel, insertamos en el vector característico la suma de los tres canales de ese píxel, entre tres: $\frac{r+g+b}{3}$. Para mejorar el diseño de nuestro código, implementamos esto dentro de la clase VectorCaracteristico. Así,

cada imagen tendrá un vector característico que tiene tamaño igual al número de píxeles de dicha imagen.

5 Generar vectores característicos con Cimg

CImg nos facilitó el proceso de generación de los vectores característicos al proveer fácil acceso a cada píxel de la imagen mediante *cimg_forXY*. Pero antes de eso, generamos el Haar Wavelet de cada imagen (con la función *haar* de CImg), le hacemos un crop a esa imagen (con la función *crop* de CImg) para solo obtener la parte de la imagen original dentro del Haar Wavelet y luego iteramos sobre cada píxel de esa *subimagen*.

El método *vectorizar* se encarga de hacer el vector característico.

La función (constructor de la clase VectorCaracteristico) saca el Haar Wavelet de la imagen y agarra una parte de la imagen generada por el Haar Wavelet.

5.1 Medidas geométricas

Experimentamos la generación del grafo con tres tipos de distancias: Euclidiana, Manhattan y Chebyshev

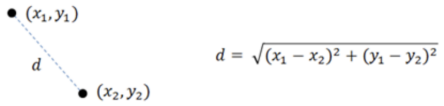

Figure 1: Distancia Euclidiana

$$|x_1 - x_2| + |y_1 - y_2|$$

Figure 2: Distancia Manhattan

$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

Figure 3: Distancia Chebyshev

6 Kruskal

Al implementar el algoritmo Kruskal buscamos encontrar el árbol de mínima expansión, ya que nuestro grafo inicial es fuertemente conexo.

6.1 Implementación

Primero, generamos todos los vectores característicos de todas las fotos. Después, hacemos un todos contra todos, sacando las distancias que hay entre todas las fotos y creamos las aristas con los vectores característicos como nodos y el la distancia como la diferencia que hay entre estos dos vectores característicos. Usamos el Fibonacci heap para poder sacar las aristas mas pequeñas, menores distancias en tiempo $\mathcal{O}(1)$ amortizado. Insertamos en el Fibonacci heap todas estas aristas, del grafo fuertemente conexo, para luego sacar el nodo mínimos hasta que tengamos un grafo conexo sin ningún ciclo.

7 Visualización

Para visualizar el grafo, usamos la herramienta *Graphviz*. Llamamos una función llamada *generatePDF* la cual describe los nodos (su nombre y imagen) y las aristas que unen dichos nodos (y su peso) y escribe dichas descripciones en un archivo el cual se lo pasaremos como parámetro al comando de graphviz para que genere el pdf.

8 Resultados

Generamos el grafo con tres distintas distancias: euclideana, manhattan y chebyshev.

Los resultados fueron los siguientes al probar la generación del grafo con las tres distancias:

Figure 4: Grafo usando Distancia Euclidiana

Como podemos observar, las diferencias entre los tres resultados, son mínimos. Las caras de las mismas personas se agrupan en los mismos componentes.

8.1 Observaciones

No pudimos experimentar con distintos cortes en el Haar Wavelet debido a que nos daba un error de ancho impar de la imagen cuando probábamos con cualquier corte mayor a 1.

Figure 5: Grafo usando Distancia Manhattan

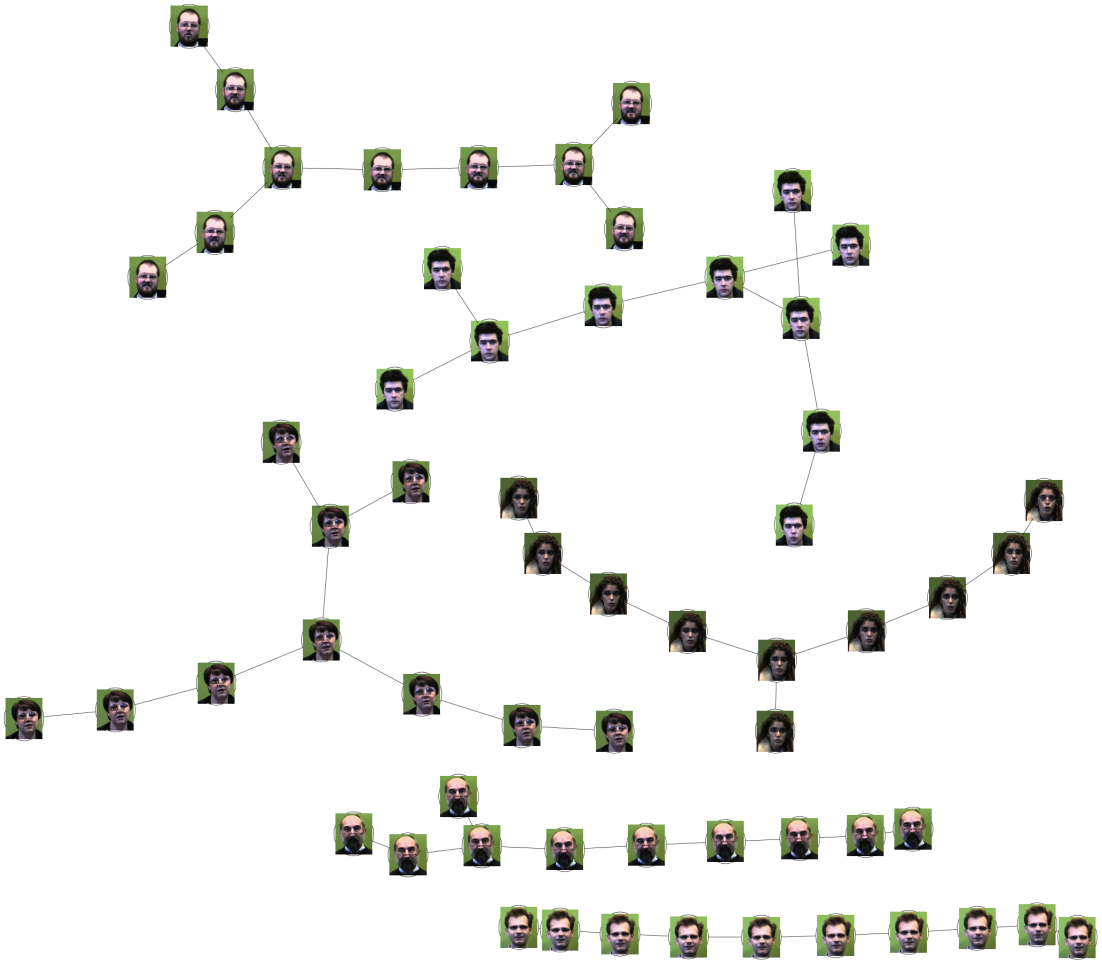

Figure 6: Grafo usando Distancia Chebyshev