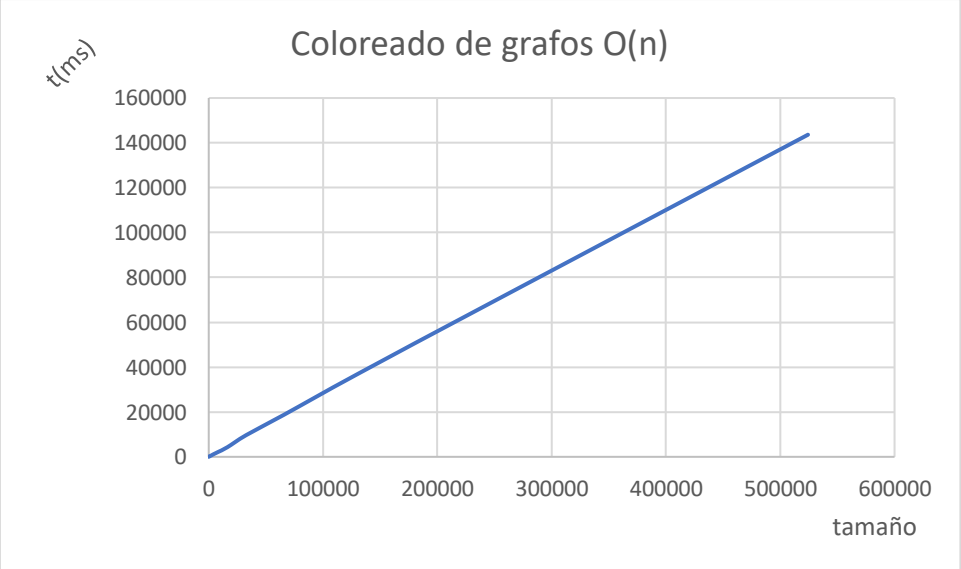
Coloreado de grafos

UO285176

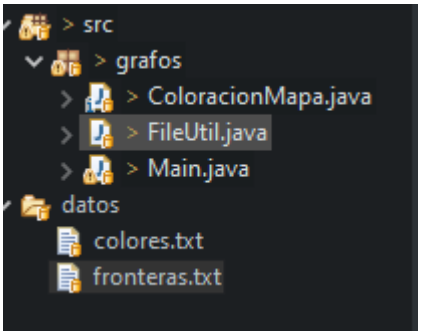
Eduardo Blanco Bielsa

- **Tabla de tiempos**

Tamaño	ms	Tiempo esperado (ms)
8	23	
16	22	46
32	29	44
64	52	58
128	77	104
256	146	154
512	192	292
1024	343	384
2048	565	686
4096	1141	1130
8192	2166	2282
16384	4322	4332
32768	9678	8644
65536	18673	19356
131072	37154	37346
262144	72808	74308
524288	143637	145616



A) Mi algoritmo se organiza en tres ficheros: ColoracionMapa.java, FileUtil y Main, así como los ficheros con los colores y las fronteras.



- La clase **FileUtil** se encarga de obtener los colores y las fronteras de los ficheros mencionados por medio de los métodos readColors y readCountries respectivamente.
- La clase **Main** crea el mapa y ejecuta el algoritmo llamando al constructor de la clase **ColoracionMapa**. Además, en ella se miden los tiempos de la tabla de tiempos anterior.
- La clase **ColoracionMapa** realiza el algoritmo. En el contructor se invocan tres métodos: sacarColores(), que obtiene los colores y los guarda en un array unidimensional, sacarPaisesYFronteras(), que obtiene los países y los adyacentes guardándolos en un HashMap y colorear() que realiza el algoritmo de coloración del mapa.

```
public ColoracionMapa() {
    sacarColores();
    sacarPaisesYFronteras();
    colorear();
}
```

- El algoritmo se basa en dos métodos:

```
private void colorear() {
    for (String pais : paises.keySet()) {
        for (String color : colores) {
            if (paises.get(pais).equals("NO")) {
                solucion.put(pais, color);
                break;
            } else if (colorUsado(pais, color)) {
                solucion.put(pais, color);
                break;
            }
        }
    }
}
```

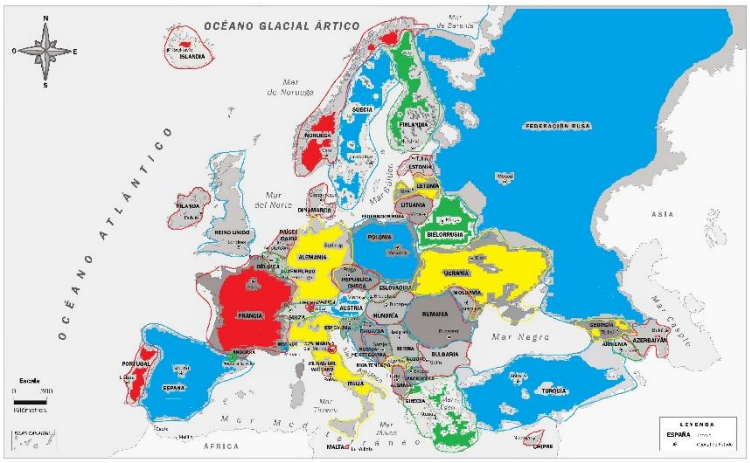
- Este método recorre las claves del HashMap de países, es decir, los países, y para cada país recorre el array de colores para asignarle el óptimo. En caso de que el país no tenga fronteras, se le asignará siempre

el primer color y se añadirá al HashMap del mapa solución. De otro modo, se llama al siguiente método privado:

```
private boolean colorUsado(String pais, String color) {
    for (String adyacente : paises.get(pais)) {
        if (solucion.get(adyacente) != null && solucion.get(adyacente).equals(color))
            return false;
    }
    return true;
}
```

- Este método busca en el HashMap de los países el país pasado como parámetro del método previo y va recorriendo sus países adyacentes o fronteras. Si en el HashMap solución el adyacente ya ha sido coloreado y tiene el color que pasamos como parámetro devuelve false. En otro caso devolverá true.

B) En mi caso han sido 4 colores. Se adjunta la siguiente imagen de la ejecución del algoritmo como **PruebaGrafos.jpg** para más detalle:



C) Sí que podría cambiar el número de colores si se usa un orden distinto.

D) 4 colores como mínimo sería la solución óptima en este mapa. Sin embargo, este número podría variar en función del mapa que intentásemos colorear.

E)

```
private void colorear() {
    for (String pais : paises.keySet()) {//O(n)
        for (String color : colores) {//O(1)
            if (paises.get(pais).equals("NO")) {
                solucion.put(pais, color);
                break;
            } else if (colorUsado(pais, color)) {//O(1)
                solucion.put(pais, color);
                break;
            }
        }
    }
}

private boolean colorUsado(String pais, String color) {//O(1)
    for (String adyacente : paises.get(pais)) {
        if (solucion.get(adyacente) != null && solucion.get(adyacente).equals(color))
            return false;
    }
    return true;
}
```

- La complejidad temporal de mi algoritmo se corresponde con O(n), ya que el bucle que recorre los países tiene un tamaño n, el bucle que recorre el array de colores tiene complejidad 1 porque son siempre 12 colores y el método privado colorUsado() tiene una complejidad 1 porque siempre recorre un número finito de países adyacentes, que es mucho menor al número de países.