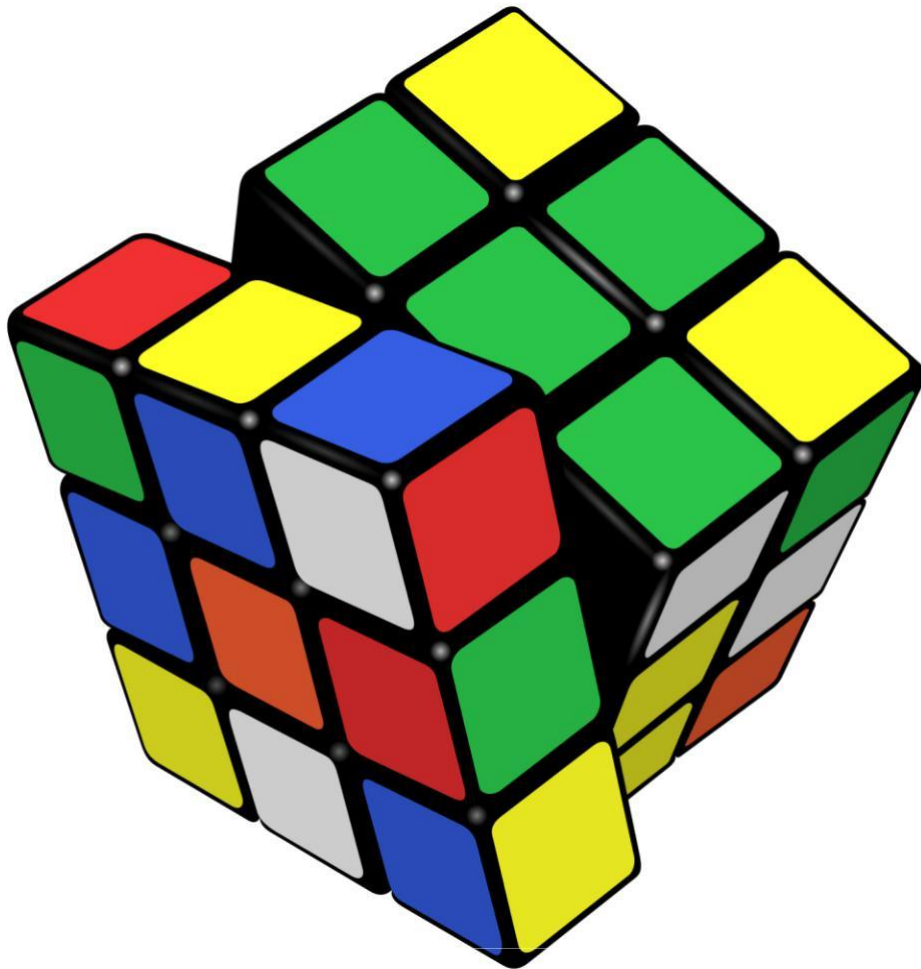


Resolución de un Cubo de Rubik con dimensiones $N \times N \times N$



Grupo EBC1-6: <https://github.com/Moniica2505/EBC1-6>

Elisa Jiménez Riaza Víctor Miguel Mora Alcázar Mónica Romero Nájera

TAREA 0: Implementación de un artefacto software N-Cubo de Rubick

Tratamiento del fichero JSON

A partir de un fichero JSON proporcionado, el programa debe extraer la configuración inicial del cubo, así como su dimensión.

Lo primero que se hace es *parsear* el fichero JSON, del que se obtienen los valores iniciales de cada cara.

El cubo tiene 6 caras: BACK, DOWN, FRONT, LEFT, RIGHT, UP.

```
public class AgentJSON {  
  
    public static Cube getCube(String ruta) throws FileNotFoundException {  
  
        byte[][] back, down, front, left, right, up;  
  
        String file = readFile(ruta);  
        JsonParser parser = new JsonParser();  
        JsonObject obj = parser.parse(file).getAsJsonObject();  
  
        back = getMatrix(obj, "BACK");  
        down = getMatrix(obj, "DOWN");  
        front = getMatrix(obj, "FRONT");  
        left = getMatrix(obj, "LEFT");  
        right = getMatrix(obj, "RIGHT");  
        up = getMatrix(obj, "UP");  
  
        return new Cube(back, down, front, left, right, up);  
    }  
}
```

Para generar un *ID* unico del estado del cubo, se pide que sea mediante *hash MD5*, para lo cual tenemos una función llamada *makeId()* en la clase Cubo, que genera el ID del cubo actual.

```
public String makeId(){  
    String sback="", sdown="", sfront="", sleft="", sright="",  
        sup=""; int n = back[0].length;  
    for(int i=0; i<n; i++){  
        for(int j=0; j<n; j++){  
            sback += back[i][j];  
            sdown += down[i][j];  
            sfront += front[i][j];  
            sleft += left[i][j];  
            sright += right[i][j];  
            sup += up[i][j];  
        }  
    }  
    return md5(sback+sdown+sfront+sleft+sright+sup);  
}
```

La función para generar el *hash MD5* es la siguiente:

```
private static String md5(String password){
    try {
        MessageDigest md = MessageDigest.getInstance("MD5");
        byte[] messageDigest = md.digest(password.getBytes());
        BigInteger no = new BigInteger(1, messageDigest);
        String hashtext = no.toString(16); while
        (hashtext.length() < 32) {
            hashtext = "0" + hashtext;
        }
        return hashtext;
    }
    catch (NoSuchAlgorithmException e) {
        throw new RuntimeException(e);
    }
}
```

Fuente: <https://www.geeksforgeeks.org/md5-hash-in-java/>

Esto va a resultar muy útil a lo largo de la práctica, sobretodo para realizar comprobaciones de resultados. Por ejemplo:

Llamar a la función: `makeId();`

Resultado: 69db38e2ce96d3044adc00e612a810b0

Representación interna del cubo

Una vez hemos obtenido la configuración del cubo mediante nuestro *Agente JSON*, construimos nuestro cubo mediante matrices bidimensionales que representan cada una de las caras.

```
public class Cube {
    private byte[][] back;
    private byte[][] down;
    private byte[][] front;
    private byte[][] left;
    private byte[][] right;
    private byte[][] up;
    private int n;

    public Cube(byte[][] back, byte[][] down, byte[][] front... etc
    { this.back = back;
      this.down = down;
      this.front = front;
      this.left = left;
      this.right = right;
      this.up = up;
      this.n = up[0].length;
    }
```

Como se puede observar, usamos la primitiva *byte* para representar las matrices bidimensionales. En un principio, usábamos otro tipo de variables como *int*, o incluso *short*, pero con las optimizaciones que hemos llevado a cabo en el desarrollo de la práctica, nos hemos decantado finalmente por esta representación.

En el proceso de construcción del cubo, hemos incluido el cálculo de la dimensión del cubo, ya que va a ser necesaria en el futuro.

Movimientos

Los movimientos se basan en 3 ejes, que llamaremos L, D y B (es una forma análoga de llamar a los ejes comúnmente denominados 'XYZ')

En la notación de los movimientos, va implícita mucha información, donde la letra corresponde al tipo de movimiento, si está en mayúscula significa que hay que rotar $+90^\circ$ y si está en minúscula, justo al contrario, -90° . Por otra parte, la letra va seguida de un número, que corresponde a la posición de la fila o columna que queremos rotar.

La forma en la que hemos planteado la implementación de los movimientos es básicamente obtener las filas, columnas o caras que hay que rotar, y posteriormente asignarlas a la nueva posición que deben ocupar. Además, en algunos casos, hay que girar una cara entera, concretamente cuando el movimiento se ejerce en los límites del cubo, por ejemplo L0 y $L_{(\text{dimensión}-1)}$

Para facilitar el proceso y mejorar la legibilidad, hacemos uso de funciones auxiliares que posteriormente usaremos dentro de la implementación de los movimientos:

```
private void girar(byte[][] matrix, boolean
    right){ byte[][] aux = clone(matrix);
    if(right)
        for(int i=0,j=matrix.length-1 ; j>-1; i++,j--)
            setCol(matrix, j, aux[i]);
    else
        for(int i=0; i<matrix.length;i++)
            setCol(matrix, i, invertir(aux[i]));

private byte[] invertir(byte[] matrix){
    byte[] result = new byte[matrix.length];
    for(int i=0,j=matrix.length-1;i<matrix.length;i++,j--)
        result[i] = matrix[j];
    return result;
}
```

Por ejemplo, mostramos como generamos el movimiento B (como es en mayúscula, el movimiento es de +90°)

```
public void B(int i){
    byte[] actual, siguiente;

    actual = up[i];
    siguiente = left[i];
    left[i] = actual;

    actual = siguiente;
    siguiente = down[i];
    down[i] = actual;

    actual = siguiente;
    siguiente = right[i];
    right[i] = actual;

    actual = siguiente;
    up[i] = actual;

    if(i==0)
        girar(back,true);
    else if(i == n-1)
        girar(front,true);
}
```

Como vemos, mediante el uso de matrices auxiliares (para no modificar el cubo en tiempo real de asignación), obtenemos y asignamos las diversas filas o columnas a las nuevas posiciones.

También vemos como se gira la cara entera correspondiente en el caso de que el movimiento se ejerza en el extremo del cubo.

A continuación, mostramos parte de otro movimiento (D), que hace uso de otras funciones como *invertir*, *getCol* o *setCol*:

```
public void D(int i){

    byte[] actual, siguiente;

    actual = front[i];
    siguiente =invertir(getCol(left, n-1-i));
    setCol(left, n-1-i, actual);

    actual = siguiente;
    siguiente =back[n-1-i];
    back[n-1-i] = actual;

    actual = siguiente;
    siguiente = invertir(getCol(right, i));
    setCol(right, i, actual);

    actual = siguiente;
    front[i] = actual;
```

Para comprobar la validez de los movimientos implementados, hacemos uso del ejemplo proporcionado, en el que se da un cubo de dimensión 10x10 con una configuración inicial. También se da el *ID MD5* tras realizar cada movimiento.

En nuestro caso, hemos implementado una clase para realizar dicha comprobación de una manera mas visual. Tras ejecutar la prueba, comprobamos que los *ID MD5* coinciden, por lo que damos por correctos los movimientos.

```
21 Tool.Pintar(cube.makeId()+" <-- ¿es igual? --> "+"69db38e2ce96d3044adc00e612a810b0 \n");
22 cube.l(3);
23 Tool.Pintar(cube.makeId()+" <-- ¿es igual? --> "+"130d0d212b8cc15f375b1b0f2cdf42ad \n");
24 cube.D(1);
25 Tool.Pintar(cube.makeId()+" <-- ¿es igual? --> "+"d83b0f604f0fbdd646497bcc400cb701 \n");
26 cube.l(1);
27 Tool.Pintar(cube.makeId()+" <-- ¿es igual? --> "+"3072cd153434334e62487aa2c52d0b1c \n");
28 cube.d(0);
29 Tool.Pintar(cube.makeId()+" <-- ¿es igual? --> "+"dab05f73e4ed15c2394f1712f9dc4fca \n");
30 cube.B(0);
31 Tool.Pintar(cube.makeId()+" <-- ¿es igual? --> "+"ff8a8cd7a7af5da72edfad5d0a801a97 \n");
32 cube.b(5);
33 Tool.Pintar(cube.makeId()+" <-- ¿es igual? --> "+"8aef8f1a6b6d427fb55581dee01e2557 \n");
34 cube.l(2);
35 Tool.Pintar(cube.makeId()+" <-- ¿es igual? --> "+"151faa80eb7b01fa8db7e8129778de10 \n");
36 cube.d(1);
37 Tool.Pintar(cube.makeId()+" <-- ¿es igual? --> "+"e8682bbb2e6fabf5971e4b471ae2d46d \n");
```

< Console Git Staging Servers

<terminated> PruebaMovimientos [Java Application] C:\Program Files\Java\jdk-12.0.2\bin\javaw.exe (12 nov. 2019 19:25:13)

```
69db38e2ce96d3044adc00e612a810b0 <-- ¿es igual? --> 69db38e2ce96d3044adc00e612a810b0

130d0d212b8cc15f375b1b0f2cdf42ad <-- ¿es igual? --> 130d0d212b8cc15f375b1b0f2cdf42ad

d83b0f604f0fbdd646497bcc400cb701 <-- ¿es igual? --> d83b0f604f0fbdd646497bcc400cb701

3072cd153434334e62487aa2c52d0b1c <-- ¿es igual? --> 3072cd153434334e62487aa2c52d0b1c

dab05f73e4ed15c2394f1712f9dc4fca <-- ¿es igual? --> dab05f73e4ed15c2394f1712f9dc4fca

ff8a8cd7a7af5da72edfad5d0a801a97 <-- ¿es igual? --> ff8a8cd7a7af5da72edfad5d0a801a97

8aef8f1a6b6d427fb55581dee01e2557 <-- ¿es igual? --> 8aef8f1a6b6d427fb55581dee01e2557

151faa80eb7b01fa8db7e8129778de10 <-- ¿es igual? --> 151faa80eb7b01fa8db7e8129778de10

e8682bbb2e6fabf5971e4b471ae2d46d <-- ¿es igual? --> e8682bbb2e6fabf5971e4b471ae2d46d
```

TAREA 1: Definición del Problema e implementación de la Frontera

Clase NodoArbol

En esta clase, definimos como estará formado un nodo de nuestro árbol, que contendrá una referencia al cubo padre (o estado anterior), el coste del camino, el estado actual, la acción que ha llevado del estado anterior al estado actual, la profundidad, y un valor aleatorio para realizar la inserción. Con esta clase, posteriormente, se realizaran pruebas de rendimiento, y se creará el método *Sucesores*.

```
public NodoArbol(NodoArbol padre, Estado est, double coste, String accion,
double d, int f){
    this.padre = padre;
    this.est = est;
    this.coste = coste;
    this.accion = accion;
    this.d = d;
    this.f = f;
}
```

De esta forma, podemos crear un nuevo nodo introduciendo los argumentos pertinentes. Por ejemplo, si queremos crear el nodo resultante de aplicar el movimiento B1 (+90°) al cubo:

```
NodoArbol nodo = new NodoArbol( padre, estado, 1, "B", 5, f);
```

Para establecer el valor que se usará para la inserción, usamos una función que devuelve un valor aleatorio (entre el rango de 1 y 10000, pero puede cambiarse a cualquier rango). Esta *f* aleatoria se usa para realizar diversas pruebas de rendimiento, posteriormente se le asignaran otros valores para una determinada estrategia

```
private static int getRandomNumberInRange(int min, int max)
{ Random r = new Random();
  return r.nextInt((max - min) + 1) + min;
}

f = getRandomNumberInRange(1, 10000);
```


Método de los Sucesores

A un objeto cubo (con su respectivo estado) se le pueden aplicar diversos tipos de movimiento, que generan cambios en el estado del cubo, como hemos visto en la Tarea 0. Concretamente existen 6 tipos básicos de movimientos (B, b, D, d, L, l), que junto con un número de movimiento, genera un movimiento específico

El método de los *Sucesores(estado)*, se encarga de generar todos los movimientos posibles aplicados al estado actual del cubo y guardarlos en una estructura de tipo *Lista*. En concreto, genera diversos estados con los distintos argumentos posibles. Hay que destacar que se usan copias del cubo para no modificar el estado original del cubo padre. A continuación, mostramos el método para generar los sucesores:

```
public List<Estado> sucesores(Estado e){
    List<Estado> result = new ArrayList<>();
    Cube cube = e.getCube();
    Cube aux;
    for(int i = 0; i < cube.getN(); i++) {
        aux = cube.clone(); aux.B(i); result.add(new Estado("B"+i,aux,1));
    }
    for(int i = 0; i < cube.getN(); i++) {
        aux = cube.clone(); aux.b(i); result.add(new Estado("b"+i,aux,1));
    }
    for(int i = 0; i < cube.getN(); i++) {
        aux = cube.clone(); aux.D(i); result.add(new Estado("D"+i,aux,1));
    }
    for(int i = 0; i < cube.getN(); i++) {
        aux = cube.clone(); aux.d(i); result.add(new Estado("d"+i,aux,1));
    }
    for(int i = 0; i < cube.getN(); i++) {
        aux = cube.clone(); aux.L(i); result.add(new Estado("L"+i,aux,1));
    }
    for(int i = 0; i < cube.getN(); i++) {
        aux = cube.clone(); aux.l(i); result.add(new Estado("l"+i,aux,1));
    }
    return result;
}
```

Por ejemplo para un cubo de 3x3, genera los estados resultantes de:

- B(0) + B(1) + B(2) + b(0) + b(1) + b(2)
- D(0) + D(1) + D(2) + d(0) + d(1) + d(2)
- L(0) + L(1) + L(2) + l(0) + l(1) + l(2)

Clase Frontera

En la frontera, se insertan los nodos generados tras expandir un nodo padre con el método de los sucesores. Se piden diversos requisitos para implementar la frontera, los más importantes son:

- Inserción ordenada en base al valor de f (ascendente)
- Probar diversas estructuras de datos y someterlas a una prueba de rendimiento, para elegir la mejor en base a los tiempos de inserción de los nodos y la cantidad de nodos que pueden insertarse (con prueba de *stress*)

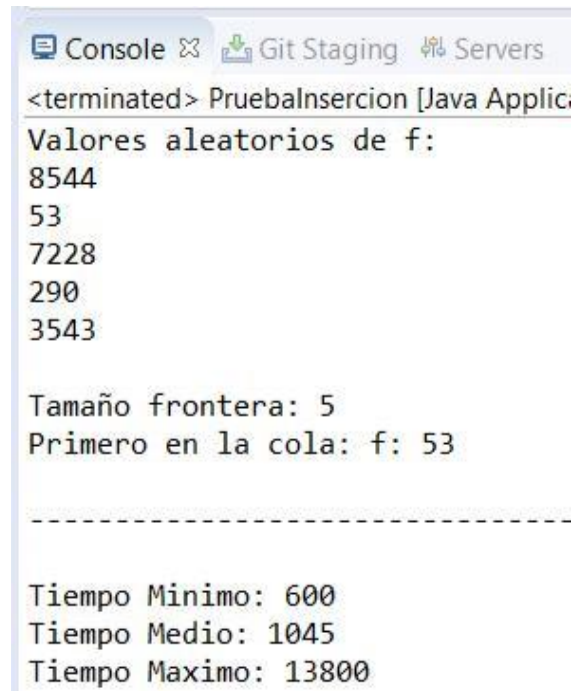
Como a lo largo de la búsqueda se van a insertar una cantidad muy grande de nodos, es importante que el tiempo no se vuelva exponencial, lo que nos obliga a evitar algunos algoritmos de ordenación como el de tipo Burbuja. De ahí la importancia de realizar una inserción ordenada. Para ello implementamos una clase llamada *PruebaInsercion*, en la que existe un método que comprueba que la inserción se realiza de forma ordenada, y otro método que calcula los tiempos de inserción mínimo, máximo y medio, de un conjunto de x inserciones (realizamos 100 inserciones)

1ª Estructura: PriorityQueue

Esta estructura, consiste en una cola, pero con una propiedad distinta, y es que junto a los objetos que se encolan, se asigna una prioridad, en nuestro caso será el valor aleatorio f el que determine la prioridad de cada nodo (posteriormente no será aleatoria). En la propia clase de *NodoArbol*, se implementa un método de comparación entre nodos para determinar la prioridad en la cola:

```
public int compareTo(Object na){
    NodoArbol nodo= ((NodoArbol)na);
    double valor = nodo.getF();
    double id = nodo.getidNodo();
    if(this.f == valor) {
        if (this.id_nodo < id){ return -1; }
        else { return 1; }
    }
    if(this.f < valor) return -1;
    else return 1;
}
```

A continuación, sometemos la estructura a la prueba de inserción, obteniendo los siguientes resultados:



```
<terminated> PruebaInsercion [Java Applic
Valores aleatorios de f:
8544
53
7228
290
3543

Tamaño frontera: 5
Primero en la cola: f: 53

-----

Tiempo Minimo: 600
Tiempo Medio: 1045
Tiempo Maximo: 13800
```

Los primeros resultados corresponden a la prueba de ordenación, como vemos, se insertan 5 nodos y posteriormente, se recupera el primer nodo de la cola, que efectivamente es el de menor f de todos.

Los segundos resultados corresponden a las pruebas de rendimiento en la inserción de nodos. Hemos elegido como medida temporal los *nanosegundos*, ya que nos ha parecido la unidad de tiempo mas equilibrada para este caso.

```
f = Tool.getRandomNumberInRange(1, 10000000);
NodoArbol nodo = new NodoArbol( inicial,estado,1,"B",5,f);
long startTime = System.nanoTime();
frontera.insertar(nodo);
long finalTime = System.nanoTime();
tiempo = finalTime-startTime;
sum = sum + tiempo;
if (tiempo < tmin) {
    tmin = tiempo;
}
if (tiempo > tmax) {
    tmax = tiempo;
}
```

Como vemos, de 100 inserciones en la frontera, el tiempo mínimo detectado es de 600 nanosegundos, el tiempo medio de todas las inserciones es de 1045 nanosegundos, y el máximo es de 13800 nanosegundos. Tras varias ejecuciones de estas pruebas, y con distinto numero de iteraciones del bucle, hemos comprobado que los valores obtenidos son siempre muy parecidos y cercanos.

2ª Estructura: **LinkedBlockingQueue**

Esta es la segunda estructura de datos que hemos elegido para comparar rendimientos, también de tipo cola.

Esta estructura, ha demostrado tener peor rendimiento a la hora de insertar valores que la cola de prioridad, y tampoco nos interesa ninguna característica especial que nos pueda beneficiar.

A continuación, mostramos los resultado de la prueba de rendimiento

```
Tiempo Minimo: 3700  
Tiempo Medio: 5099  
Tiempo Maximo: 12700
```

La mayor variación corresponde al tiempo mínimo (que es notablemente mayor, en todas las ejecuciones que hemos realizado). Esta subida del tiempo mínimo, arrastra el tiempo medio, que también crece. En cuanto al tiempo máximo, observamos que suele ser muy parecido en ambas estructuras.

Tras estas pruebas, determinamos que la estructura elegida para implementar nuestra frontera será la **PriorityQueue**.

Prueba de Stress

El siguiente paso, es someter nuestra frontera a una prueba de *Stress*, en la que se insertan nodos del tipo *NodoArbol* de forma ininterrumpida, es decir, mediante un bucle infinito.

El objetivo de esta prueba es conocer la “capacidad de aguante” de nuestra frontera, y de esta manera saber que tipo de excepción vamos a tener que capturar y controlar.

```
while(true) {  
    f = Tool.getRandomNumberInRange(1, 100000000);  
    NodoArbol nodo = new NodoArbol( inicial,estado,1,"B",5,f);  
    frontera.insertar(nodo);  
    Tool.Pintar(frontera.size());  
}
```

Ejecutamos la prueba de 2 formas distintas, una es mostrando el tamaño de la frontera para saber que tamaño tiene al saltar la excepción, y otra sin calcularlo, que es mas rápido y clarificador.

A screenshot of a Java IDE's console window. The window has tabs for 'Console', 'Git Staging', and 'Servers'. The console output shows a terminated Java application with a stack trace for a 'java.lang.OutOfMemoryError: Java heap space'. The stack trace includes the following lines: 'at java.base/java.util.Arrays.copyOf(Array.java:3721)', 'at java.base/java.util.Arrays.copyOf(Array.java:3690)', 'at java.base/java.util.PriorityQueue.grow(PriorityQueue.java:306)', 'at java.base/java.util.PriorityQueue.offer(PriorityQueue.java:345)', 'at dominio.FronteraOrdenada.insertar(FronteraOrdenada.java:17)', and 'at presentacion.Stress.main(Stress.java:30)'.

```
<terminated> Stress [Java Application] C:\Program Files\Java\jdk-12.0.2\bin\javaw.exe (12 nov. 2019 14:20:55)  
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  
    at java.base/java.util.Arrays.copyOf(Array.java:3721)  
    at java.base/java.util.Arrays.copyOf(Array.java:3690)  
    at java.base/java.util.PriorityQueue.grow(PriorityQueue.java:306)  
    at java.base/java.util.PriorityQueue.offer(PriorityQueue.java:345)  
    at dominio.FronteraOrdenada.insertar(FronteraOrdenada.java:17)  
    at presentacion.Stress.main(Stress.java:30)
```

Como era previsible, la excepción tiene que ver con la falta de memoria. A continuación, ejecutamos pero calculando el tamaño máximo de la frontera, para saber cuantos nodos soporta (29181991)

```
29181991  
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space  
    at java.base/java.util.Arrays.copyOf(Array.java:3721)  
    at java.base/java.util.Arrays.copyOf(Array.java:3690)  
    at java.base/java.util.PriorityQueue.grow(PriorityQueue.java:306)  
    at java.base/java.util.PriorityQueue.offer(PriorityQueue.java:345)  
    at dominio.FronteraOrdenada.insertar(FronteraOrdenada.java:17)  
    at presentacion.Stress.main(Stress.java:30)
```

TAREA 2: Estrategias para Algoritmo básico de búsqueda

Introducción

En esta tarea, se pide implementar diversos algoritmos de búsqueda para satisfacer distintas estrategias. También se pide implementar un sistema de poda, para el que se hará uso de una estructura de datos capaz de almacenar una gran cantidad de nodos visitados, referenciados por sus *hash MD5*.

Método para calcular la f

Como sabemos, cada estrategia tiene una f distinta, en concreto:

- Anchura: $f = profundidad$
- Coste Uniforme: $f = coste$
- Profundidad: $f = -profundidad$
- A*: $f = coste + heurística$
- Voraz: $f = heurística$

```
private static double calcularF(NodoArbol padre, Estado estado, String
estrategia){
    double g=0, h=0;

    switch(estrategia){
        case "anchura":
            g = padre.getD()+1;
            break;
        case "costo uniforme":
            g =
            padre.getCoste()+1; break;
        case "profundidad":
            g = -(padre.getD()+1);
            break;
        case "voraz":
            h = estado.getCube().getH();
            break;
        case "A":
            g = padre.getD()+1;
            h = estado.getCube().getH();
            break;
    }
    return (g+h);
}
```

Métodos para realizar las búsquedas

Para calcular las búsquedas según el tipo de estrategia son necesarios dos métodos, que se definen a continuación:

Búsqueda(String estrategia,int Prof_Max,int Inc_Prof)

```
public boolean busqueda(String estrategia,int Prof_Max,int Inc_Prof)
throws Exception {
    solucion = null;

    switch(estrategia) {
        case "anchura":
            solucion = busquedaAcotada(estrategia,Prof_Max);
            break;
        case "costo uniforme":
            solucion = busquedaAcotada(estrategia,Prof_Max);
            break;
        case "profundidad":
            solucion = busquedaAcotada(estrategia,Prof_Max);
            break;
        case "voraz":
            solucion = busquedaAcotada(estrategia,Prof_Max);
            break;
        case "A":
            solucion = busquedaAcotada(estrategia,Prof_Max);
            break;
    }
    Tool.Pintar(solucion.getEstado());
    Tool.Pintar(solucion);
    if(solucion!=null) {
        save("rubick.out",solucion.toString());
        return true;
    }
    return false;
}
```

Como se ve en el método anterior, se empieza creando una búsqueda. Hacemos uso de un *switch* para elegir la estrategia correspondiente. Después se invoca al método “búsqueda acotada”, (que explicaremos a continuación).

Cuando acaba, pintamos el estado del nodo objetivo (configuración del cubo resuelto) y la solución (secuencia de movimientos, con sus respectivos estados, profundidad, valor de f y otros atributos que forman el camino desde el nodo inicial al nodo objetivo). Además, como se pide en el enunciado, guardamos en un fichero la solución encontrada al problema.

BúsquedaAcotada(String estrategia,int Prof_Max)

```
private NodoArbol busquedaAcotada(String estrategia,int Prof_Max) {
    boolean solucion = false;
    NodoArbol n_actual = null;

    if(estadoInicial != null){
        Frontera frontera = new
        FronteraOrdenada(); cnt_creado = 1;
        cnt_frontera = cnt_abierto = 0;

        n_actual= new NodoArbol(null, estadoInicial, "",0, 0, 0);
        frontera.insertar(n_actual);
        cnt_frontera++;

        while(!solucion && !frontera.esVacia()){
            n_actual = frontera.eliminar();
            cnt_abierto++;

            if(esObjetivo(n_actual.getEstado())) {
                solucion = true;
            } else
                try{
                    List<Estado> LS = espacioEstados.sucesores(n_actual.getEstado());
                    List<NodoArbol> LN = ListaNodosArbol(LS,n_actual,Prof_Max,estrategia);
                    cnt_frontera += LN.size();
                    frontera.insertarLista(LN);
                }catch(Exception e){
                }
            }
        }
    }
    return n_actual;
}
```

Este método es el “eje central” del algoritmo de búsqueda. Como se ve, es un método que devuelve un objeto del tipo *NodoArbol* (objetivo).

Primero, empezamos inicializando la frontera, y otras variables que usaremos en la función. Posteriormente, a partir del *estadoInicial* del cubo, creamos el primero nodo y lo insertamos en la frontera.

**nota al profesor:* Las variables *cnt_creado*, *cnt_frontera* y *cnt_abierto*, las usamos nosotros para realizar pruebas, y saber cuantos nodos se expanden, cuantos se insertan, etc... También las usamos para establecer el ID de los nodos, pero la función que tienen es ayudarnos, no tienen mas relevancia

A continuación, comenzamos el bucle de búsqueda:

- 1- Sacamos un nodo de la frontera (se insertaban según su valor *f*)
- 2- Comprobamos si el nodo es el objetivo con la función *esObjetivo()*

```
private boolean isEqual(byte[][] matrix)
{ byte aux = matrix[0][0];
  for(int i=0; i< matrix.length;i++)
      for(Byte b :matrix[i])
          if(!b.equals(aux))
              return false;
  return true;
}
```



```

public boolean esObjetivo(Estado e) {
    Cube cube = e.getCube();
    return isEqual(cube.getBack())
        && isEqual(cube.getDown())
        && isEqual(cube.getFront())
        && isEqual(cube.getLeft())
        && isEqual(cube.getRight())
        && isEqual(cube.getUp());
}

```

3- Si es el objetivo, cambiamos el *boolean* solución a *true* y salimos del bucle de búsqueda.

4- Si no es el nodo objetivo, lo expandimos, creando una lista con los respectivos sucesores, llamada *LS*

5- A partir de la lista *LS*, creamos la lista *LN*, que va a contener los nodos creados a partir de los sucesores

```

private static List<NodoArbol> ListaNodosArbol(List<Estado> LS, NodoArbol
n_padre, int Prof_Max, String estrategia){
    List<NodoArbol> result = new LinkedList<>();

    for(Estado estadoActual : LS) {
        double f = calcularF(n_padre, estadoActual, estrategia);

        if(n_padre.getD() < Prof_Max) {
            NodoArbol nodo = new NodoArbol(n_padre, estadoActual,
            estadoActual.getAcci(), n_padre.getCoste()+1, n_padre.getD()+1, f);
            cnt_creado++;
            nodo.setId_nodo(cnt_creado);
            result.add(nodo);
        }
    }
    return result;
}

```

Como se ve, mientras vaciamos la lista de los sucesores, calculamos los respectivos valores de *f*. A partir de ahí, ya podemos crear los nodos, pues tenemos todos los atributos necesarios. Asignamos el ID al nodo con nuestro contador, y por ultimo lo añadimos a la lista *LN*

6- Insertamos en la frontera los nodos creados anteriormente. Como ya lo hicimos antes en la memoria, no volveremos a explicar el funcionamiento ni la razón de ser de la frontera.

A partir de este punto, el bucle vuelve a realizar otra iteración, siguiendo los mismos pasos: saca un nodo de la frontera, comprueba si es objetivo, expande.... etc

Método para podar los nodos

Este método es el encargado de quitar todos los nodos cuyos estados han sido visitados con anterioridad y han obtenido mejor resultado en su valor de f que el actual.

Hemos implementado una *LinkedList* que contiene todos los nodos que están o han pasado por la frontera, tanto si han sido explorados como si no, para poder comparar un nodo concreto con todos los nodos de la lista. En caso de encontrar un nodo con el mismo estado cuya profundidad sea superior al nodo encontrado, se cambia.

Nos vemos obligados a admitir, que si bien hemos hecho diversas pruebas con la poda, y hemos explorado varias opciones, en esta versión de la practica no lo hemos llegado a implementar en nuestro Main principal, es decir, no estará la opción de realizar búsquedas con poda. Seguimos trabajando para pulir errores e incluirla pronto.

TAREA 3: Estrategias A* y voraz

Para implementar estas dos estrategias es necesario calcular una heurística en base a la fórmula de entropía facilitada por el profesor.

Método de la entropía

Primero de todo, traducimos la fórmula facilitada a lenguaje Java. Este lenguaje no contiene ningún método para calcular un logaritmo de una base concreta, que no sea el numero e o base 10, por lo que hemos hecho la relación: $\log(\text{número})/\log(\text{base deseada}) = \log \text{ en base deseada}$

```
private double entropia(byte[][] matrix)
{
    double aux;
    int[] contador = new int[6];
    double entropia=0;

    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            contador[matrix[i][j]]++;

    for (int c= 0; c<6; c++) {
        if (contador[c]>0.0) {
            aux = (contador[c]*1.0)/(n*n);
            entropia = entropia + aux * (Math.Log(aux)/Math.Log(6));
        }
    }
    return -entropia;
}
```

Lo que hace el método, es contar cuantas casillas de un color concreto hay en una determinada cara. Posteriormente, mediante un bucle, se calcula el desorden de la cara y se retorna.

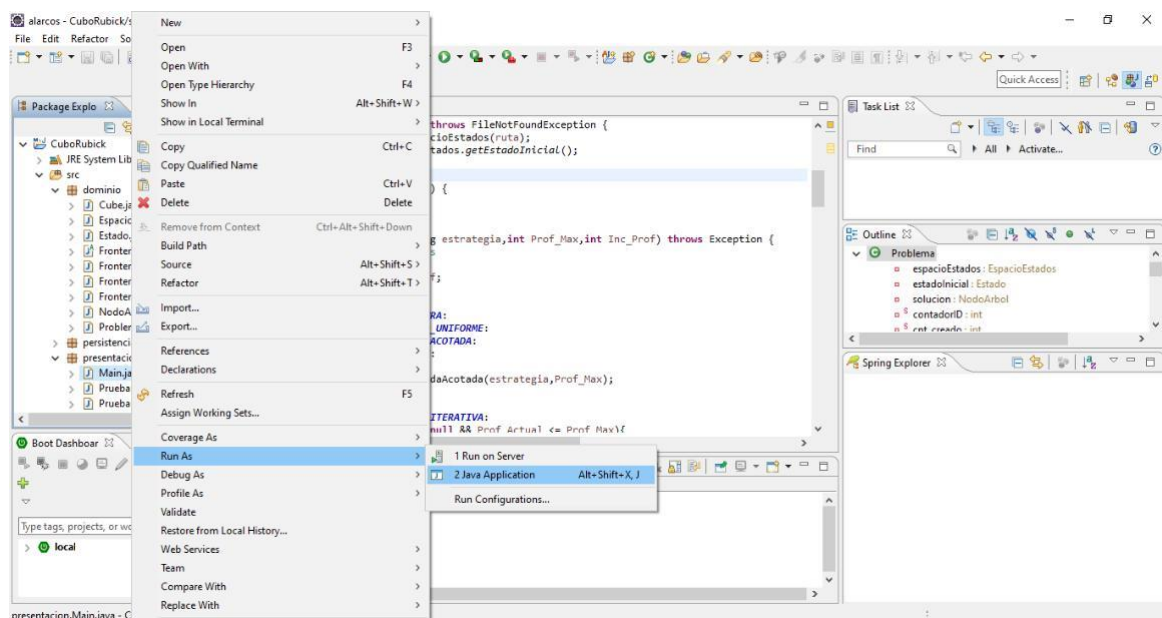
El método anterior, calcula la entropía de una sola cara, pero como queremos saber la de todo el cubo, sumamos las entropías obtenidas

```
public double getH() {  
    return entropia(back) + entropia(down) + entropia(front)  
    + entropia(left) + entropia(right) + entropia(up); }  
}
```

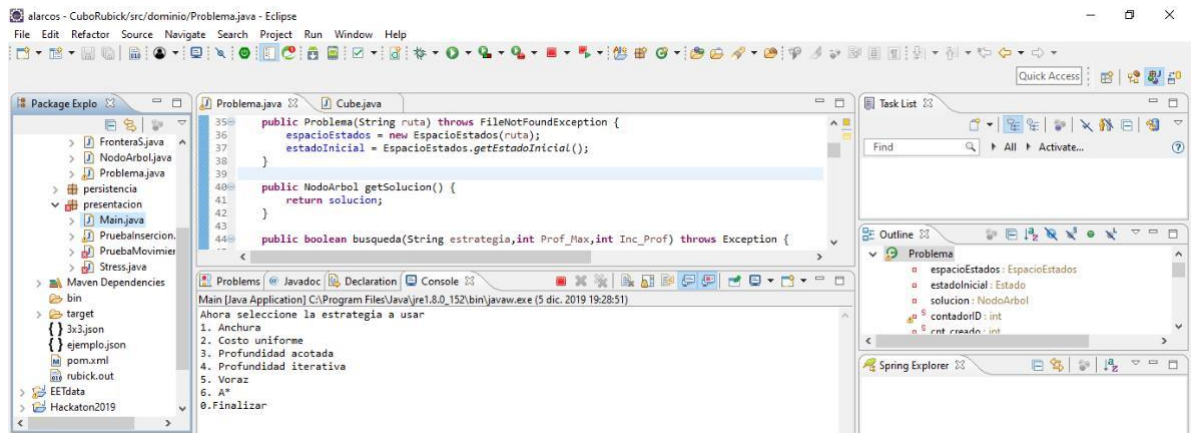
Con estas funciones, a la hora de calcular la f (como vimos anteriormente en el método), ya podemos obtener la heurística en estas dos estrategias, ya que en el caso de la Voraz, su f corresponde únicamente con la entropía, y en el caso de A^* , a la heurística tan solo tenemos que sumarle el coste.

Manual de usuario

1. Ejecutar la aplicación desde Eclipse (el *Main* principal se encuentra en el paquete *Presentación*)



2. El programa mostrará una pantalla donde nos pedirá la ruta del archivo que contiene el cubo desordenado.
3. Una vez introducido, aparecerá una lista de opciones, donde habrá que elegir el tipo de búsqueda que queramos utilizar.



4. Poner el numero de la búsqueda deseada y pulsar ENTER.
5. Introducir la profundidad máxima.

Si el problema tiene solución, nos la mostrara por pantalla y además la guardara en un archivo solución.out, si por el contrario no encuentra solución, también nos avisara de esto con un mensaje en la pantalla.

Opinión personal

Elisa: Esta práctica me ha resultado interesante para poder asentar los conocimientos teóricos de la asignatura, aunque me ha resultado un poco complicado poder seguir la traza de los caminos obtenidos y sobre todo plantear los movimientos ya que no es tan visible como un grafo de un mapa, por ejemplo. En general me ha parecido útil para tener una base de conocimientos de inteligencia artificial.

Victor: Esta práctica me ha parecido realmente interesante, y bien planteada para entender los conceptos básicos de búsquedas. También me han llamado especialmente la atención otras tareas necesarias en el desarrollo de la práctica. Por ejemplo, las pruebas de rendimiento y de *stress* para comparar estructuras de datos y elegir la mejor frontera, la implementación de los movimientos, o profundizar en el manejo del concepto *hash MD5* o *JSON*, ya que nunca lo había usado en la práctica. Todo esto me han ayudado a adquirir nuevos conocimientos y experiencias. También he aprendido a conocer y entender mejor los “límites” de mi ordenador, que de por sí no parece muy llamativo, pero extrapolando estas experiencias a otros campos de la informática, he comprendido mejor muchos factores que antes no tenía tan claros.

Mónica: