Documentación Práctica Sistemas Inteligentes

Artefacto del cubo creado

En primer lugar se ha creado en un archivo JSON un cubo inicialmente desordenado, es decir, con diferentes colores en cada una de sus caras. En el archivo JSON se han creado 6 matrices que representan cada uno de los lados del cubo que contienen unos valores determinados que representan cada uno de los colores, siendo:

1-Rojo
2-Azul
3-Amarillo
4-Verde
5-Naranja
6-Blanco

El código del archivo JSON con el artefacto del cubo que se ha subido a Github es el siguiente:

Idea general de los movimientos

Los movimientos del cubo en un principio se implementan en métodos que reciben como parámetro una matriz con cada uno de los colores que tiene cada casilla del cubo y un entero que representa el tipo de movimiento a realizar. En función de una determinada condición se realizará un movimiento u otro. Por ejemplo, si queremos girar hacia abajo (GirarDown), en caso de cumplir con la condición principal, se realiza el movimiento hacia abajo pero si no es así el movimiento que se realiza es hacia arriba. En resumen, esta condición maneja la posibilidad de que se realice el movimiento que se desea o de que sea su contrario en lugar de éste.

Participantes

Los participantes de este grupo de trabajo (Grupo EB1-3) son:
-Arturo Alcañiz Guijarro
-Juan Antonio Lozano Núñez
-David Illescas Herrera

Repositorio Github

Tanto el código de la práctica como la documentación asociada a la misma se almacenará en un repositorio de Github. La dirección de nuestro repositorio es la siguiente:
https://github.com/juanan280/EB1-3

Tarea 1

En esta tarea se pedía definir un espacio de estados, el problema de la generación de los movimientos del Cubo de Rubick y un árbol de busqueda.

Clase NodoArbol

Se ha creado una clase específica llamada NodoArbol que contiene los atributos que debe tener cada nodo del árbol de búsqueda.

```java

public class NodoArbol {

    NodoArbol padre;
    String estado;
    String accion;
    int f;

    public NodoArbol(NodoArbol padre, String estado, String accion, int f) {
        this.padre = padre;
        this.estado = estado;
        this.accion = accion;
        this.f = f;
    }

    public NodoArbol(String estado, int f) {
        this.estado = estado;
        this.f = f;
    }

    public NodoArbol getPadre() {
        return padre;
    }

    public void setPadre(NodoArbol padre) {
```

```java
    public void setPadre(NodoArbol padre) {
        this.padre = padre;
    }

    public String getEstado() {
        return estado;
    }

    public void setEstado(String estado) {
        this.estado = estado;
    }

    public String getAccion() {
        return accion;
    }

    public void setAccion(String accion) {
        this.accion = accion;
    }

    public int getf() {
        return f;
    }

    public void setf(int f) {
```

```
49
50      public void setf(int f) {
51          this.f = f;
52      }
53  }
```

Como bien puede verse en el código de arriba en la clase NodoArbol se han declarado cuatro atributos que son:
-La información de acceso al padre que es de tipo NodoArbol, ya que el padre es otro nodo que forma parte de ese mismo árbol de búsqueda.
-El estado como un String.
-La acción también como un String como el estado anterior que nos indicará el tipo de movimiento a realizar para llegar desde el padre a ese estado actual(nodo actual).
-La frontera como un valor entero que luego en el método GenerarFrontera y encontrarSolución serán valores aleatorios.

Cabe destacar que existen dos constructores para esta clase, cuyo objetivo es diferenciar dos tipos de instancias diferentes, es decir, el constructor con los 4 atributos pertenece al nodo actual del árbol de búsqueda y el que solo tiene 2 atributos (estado, frontera) pertenece al nodo padre. Los demás métodos que se han implementado en esta clase son getters y setters de todos los atributos declarados.

Clase Frontera

```
29
30              while(iter.hasNext()) {
31                  aux = iter.next().getf();
32                  if(f<aux) {
33                      break;
34                  }
35                  posicion++;
36              }
37          }
38          return posicion;
39      }
40
41      public void eliminar() {
42          nodos.remove(0);
43      }
44
45      public boolean estavacia() {
46          if(nodos == null || nodos.isEmpty()) {
47              return true;
48          }else {
49              return false;
50          }
51      }
52
53      public String toString() {
54          Iterator<NodoArbol> iter = nodos.iterator();
```

```
 4  import java.util.ArrayList;
 5  import java.util.Iterator;
 6
 7  public class Frontera {
 8
 9      ArrayList<NodoArbol> nodos = new ArrayList<NodoArbol>();
10
11      public void insertar(NodoArbol nodo) {
12          int posicion = obtenerPosicionInsercion(nodo.getf());
13          if(posicion>(nodos.size()-1)) {
14              nodos.add(nodo);
15          }
16          else {
17              nodos.add(posicion, nodo);
18          }
19      }
20
21      public int obtenerPosicionInsercion(int f) {
22          int posicion = 0;
23          int aux = 0;
24          if(estavacia()) {
25              posicion = 0;
26          }else {
27
28              Iterator<NodoArbol> iter = nodos.iterator();
29
```

```
55          int posicion = 0;
56          String texto = "";
57          while(iter.hasNext()) {
58              NodoArbol a = iter.next();
59              texto = texto+"Nodo frontera en posicion: "+posicion+", estado: "+a.getEstado()+", accion: "+a.getAccion()+"
60              texto = texto+"\n";
61              posicion++;
62          }
63          return texto;
64      }
65
66      public ArrayList<NodoArbol> getNodos(){
67          return nodos;
68      }
69
70  }
```

Para esta clase solo declaramos una lista con todos los nodos que formarán parte de la solución mientras no lleguemos a la frontera.

Se ha creado un método getNodos() con el objetivo de dar la secuencia de nodos que forman parte de la solución, ya que se debe tener en cuenta que la solución no es solamente el estado final, sino la secuencia de nodos que se ha debido seguir para llegar a ese estado final.

Para saber en qué posición de la lista de nodos debemos insertar un nuevo nodo que puede formar parte de la solución, se ha utilizado el método iterator() de la clase Iterator en el que mientras la frontera tenga el mismo o un número mayor de elementos de los que se obtienen del iterator, se vaya incrementando en uno a la posición en la que se encontraba en ese momento el iterator mientras recorre la lista de nodos. Así finalmente se encontrará una posición para insertar.

También se controla el hecho de que la lista de nodos se encuentre vacía mediante un método de tipo boolean que puede devolvernos true o false.

El método insertar() funciona de una manera sencilla, puesto que si el valor de la posición en la que queremos insertar el nodo es mayor que la última posición, entonces este nodo se insertará en la posición siguiente a la última de antes. Si no es así, el nodo se insertará en la posición que se le indique al método add().

```
41    public static ArrayList<String> encontrarSolucion(String file, int limiteProfundidad) {
42        Random rd = new Random();
43        ArrayList<String> solucion = new ArrayList<String>();
44        ArrayList<String> visitados = new ArrayList<String>();
45        NodoArbol a = null;
46        Frontera f = new Frontera();
47        Object datosCubo[];
48        try {
49            String cubo = leerjson(file);
50            NodoArbol padre = new NodoArbol(cubo, rd.nextInt(100));
51            f.insertar(padre); // La frontera inicial es el cubo del que partimos
52
53            while (f.getNodos().size()>=1) {
54
55                a = f.getNodos().get(0);
56
57                if(EsObjetivo(a.getEstado())) {
58                    break;
59                }else {
60
61                    if(calcularDepth(a)<limiteProfundidad) {  //Control de profundidad
62
63                        String md5 = DigestUtils.md5Hex(a.getEstado());
64                        visitados.add(md5);
```

```
79            e.printStackTrace();
80        } catch (IOException e) {
81            e.printStackTrace();
82        } catch (ParseException e) {
83            e.printStackTrace();
84        }
85        if(EsObjetivo(a.getEstado())) {
86
87            while(a.getPadre() != null) {
88
89                solucion.add(0, a.getAccion());
90                a = a.getPadre();
91
92            }
93
94            return solucion;
95
96        }else {
97
98            return null;
99
100       }
101
102   }
```

```
64                        visitados.add(md5);
65                        f.eliminar();
66                        GenerarFrontera(visitados, f, a);
67
68                    }else {
69
70                        f.eliminar();
71
72                    }
73                }
74            }
75
76
77
78        } catch (FileNotFoundException e) {
```

En esta función de la clase principal se ha implementado el método que se encarga de encontrar la solución mientras que se encuentre el fichero json del cubo (Uso de control de excepciones para la abertura de un fichero mediante sentencias del tipo try-catch). Básicamente mientras no se cumpla la función objetivo y no se llegue a la profundidad máxima del árbol de búsqueda, se procederá a ir generando poco a poco la frontera y a visitar cada uno de los posibles nodo solución e ir metiendo y sacando de la frontera dependiendo de si ese nodo forma parte de la solución.

```java
public static boolean EsObjetivo(String estado) {

    ArrayList<String[][]> cubo = new ArrayList<String[][]>();
    inicializarcubomatriz(cubo, estado);

    boolean comprobar = true;
    int ca = 0;
    int i = 0;
    int j = 0;

    while(comprobar == true && ca<6) {

        i = 0;

        String[][] cara = cubo.get(ca);

        String color = cara[0][0];

        while(comprobar == true && i<cara.length) {


            j=0;

            while(comprobar == true && j<cara[i].length) {

                if(Integer.parseInt(cara[i][j]) != Integer.parseInt(color)) {
                    comprobar = false;

                }

                j++;
            }
            i++;
        }
        ca++;
    }

    return comprobar;

}
```

Con esta función comprobamos que el estado final en el que se encuentre el cubo es igual que la función objetivo, es decir, que los colores en una cara son iguales (Ej: en una cara solo hay colores rojo) y no hay al menos un color diferente. En caso de que así sea, no habremos llegado aún a la función objetivo.

```java
public static int calcularDepth(NodoArbol a) {
    int d = 0;
    while(a.getPadre() != null) {
        a = a.getPadre();
        d++;
    }
    return d;
}
```

Con esta función calculamos la profundidad total que tiene el árbol de búsqueda.

En la función GenerarFrontera(), le pasamos como entrada un ArrayList que contendrá todos los nodos que se vayan visitando, la frontera que queremos calcular y el nodo padre con el objetivo de que al final vayamos insertando en ella cada uno de los nodos del árbol de búsqueda no visitados teniendo en cuenta que cada cara cuenta con su propio árbol de búsqueda. Por último, la frontera será un objeto de tipo frontera.

Tarea 2

Hemos creado el algoritmo básico de búsqueda necesario para implementar las diferentes estrategias con dos metodos el de Busqueda y ListaNodos.

Cambios en la clase principal:

Método Busqueda:

Para comenzar se crea 3 ArrayLists para guardar los nodos de la solución, para guardar los ids de los nodos visitados y otro para guardar ids.
Después se crea la frontera vacia. Se lee el cubo inicial y se introduce en la frontera.
Tras esto comienza el bucle hasta que se encuentre un estado que sea la solucion o hasta que se vacie la frontera. Se saca un nodo de la frontera si es el nodo objetivo se introduzen en el array solucion los nodos por los que hay que pasar para llegar a la solución

```
public static ArrayList<ArrayList> Busqueda(EspacioEstados prob, String estrategia, int limiteProfundidad) {

    ArrayList<NodoArbol> solucion = new ArrayList<NodoArbol>();
    ArrayList<String> visitados = new ArrayList<String>();
    ArrayList<String> ids = new ArrayList<String>();

    Frontera f = new Frontera();

    Random rd = new Random();

    boolean sol = false;

    NodoArbol a = null;

    int n=0;

    try {

        String cubo = LecturaJSON.leer(prob.getFile());

        NodoArbol padre = new NodoArbol(cubo, 0, 0);

        f.insertar(padre); // La frontera inicial es el cubo del que partimos
```

```
while (sol == false && !f.estavacia()) {

    a = f.getNodos().get(0);

    if(EsObjetivo(a.getEstado())) {

        visitados.add(DigestUtils.md5Hex(a.getEstado()));

        while(a.getPadre() != null) {

            ids.add(0, ""+visitados.indexOf(DigestUtils.md5Hex(a.getEstado())));
            solucion.add(0, a);
            a = a.getPadre();

        }

        ids.add(0, ""+visitados.indexOf(DigestUtils.md5Hex(a.getEstado())));
        solucion.add(0, a);

        sol = true;
        break;
```

Si el nodo sacado de la frontera no es el nodo objetivo se introducen en el array de visitados y después se buscan sus sucesores con el método que lleva ese mismo nombre para después con ListaNodos crea ¡r una lista de nodos que se introduzcan en la frontera, si ya han sido visitados se eliminan de la frontera.
También se tiene en cuenta el límite de la profundidad ya que si el nodo tiene una profundidad mayor se elimina de la frontera y terminaría su seguimiento.
Para terminar, se devuelve un array con la solución si esta se ha encontrado.

```
}else {

    if(a.getd()<limiteProfundidad) {  //Control de profundidad

        ArrayList<String[]> sucesores = prob.Sucesores(a.getEstado());

        ArrayList<NodoArbol> ListaNodos = ListaNodos(sucesores, a, estrategia);

        visitados.add(DigestUtils.md5Hex(a.getEstado()));
        f.eliminar();

        for(int i=0; i<ListaNodos.size(); i++) {

            String md5sucesor = DigestUtils.md5Hex(ListaNodos.get(i).getEstado());


            if(!visitados.contains(md5sucesor)) {
                NodoArbol s = ListaNodos.get(i);
                NodoArbol nodofrontera = new NodoArbol(s.getPadre(), s.getEstado(), s.getAccion(), s.getf(),

                f.insertar(nodofrontera);
            }
        }
```

```
                }else {

                    f.eliminar();
                }

            }
        }



    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ParseException e) {
        e.printStackTrace();
    }

    ArrayList<ArrayList> SOL = new ArrayList<ArrayList>();
    SOL.add(solucion);
    SOL.add(0, ids);
    return SOL;


}
```

Este método crea una lista de nodos con todos sus atributos dependiendo de la estrategia seleccionada, cambiando la f dependiendo de esa estrategia.
Método ListaNodos:

```
public static ArrayList<NodoArbol> ListaNodos(ArrayList<String[]> ListaSucesores, NodoArbol nodo_actual, String

    ArrayList<NodoArbol> ListaNodos = new ArrayList<NodoArbol>();
    Random rd = new Random();
    int f = 0;

    for(int i=0; i<ListaSucesores.size(); i++) {

        String[] estado_sucesor = ListaSucesores.get(i);

        int Aleatorio = rd.nextInt(100);

        if(estrategia.equals("anchura") || estrategia.equals("coste_uniforme")) {

            f = nodo_actual.getd()+1;

        }else if(estrategia.equals("profundidad")) {

            f = 1/(nodo_actual.getd()+2);

        }

        NodoArbol a = new NodoArbol(nodo_actual, estado_sucesor[0], estado_sucesor[1], f, nodo_actual.getd()+1)
        ListaNodos.add(a);

    }

    return ListaNodos;
```