SISTEMAS INTELIGENTES

Memoria de prácticas

Rodrigo Díaz - Hellín Valera

Ciudad Real, 18 de Diciembre de 2016

TABLA DE CONTENIDOS

HITO 1

---

1 - Introducción

El objetivo del trabajo de prácticas consiste en la implementación de un programa que resuelva gráficamente un problema de un puzzle de dimensiones ((NxM)-1) a partir de una imagen en formato .png.

En el hito 1 se llevarán a cabo la definición e implementación del artefacto puzzle y del conjunto de operaciones básicas sobre éste. Estas operaciones son:

1. Creación: Construcción de un artefacto puzzle de dimensiones N columnas y M filas a partir de un fichero gráfico con extensión .png donde la pieza del puzzle que corresponde inicialmente a la posición 0x0 (esquina superior-izquierda) se trata de la pieza vacía y será representada por una imagen en negro.

2. Generación de la imagen: Una vez creado un artefacto puzzle se deberá poder generar y almacenar una imagen del puzzle como un fichero gráfico con extensión .png.

3. Reconstrucción: Construcción de un objeto puzzle a partir de una imagen del puzzle desordenado. Esta operación deberá comparar las imágenes correspondientes a cada una de las piezas para garantizar si existe relación entre el puzzle original y el modificado.

4. Movimientos válidos: Deberán definirse los movimientos válidos que pueden realizarse sobre el puzzle en todos los posibles estados del mismo.

5. Movimientos: Implementación de los movimientos de desplazamiento sobre las piezas del puzzle.

2 - Tecnologías y librerías utilizadas

El lenguaje de programación utilizado para la implementación del artefacto y operaciones básicas del puzzle ha sido JAVA, debido a la experiencia en el uso de este lenguaje y a las librerías que proporciona como *java.awt* y *javax.imageio* disponibles en la API de Java en https://docs.oracle.com/javase/7/docs/api/ y necesarias para la manipulación y comparación de imágenes.

Aunque por el momento no se ha hecho referencia a aspectos relativos al diseño ni a la implementación del programa, hemos querido mostrar aquí el siguiente gráfico que permite conocer el orden de complejidad del problema planteado mediante la representación del tiempo de respuesta en función del tamaño del problema

Como se observa en la *Figura 1*, la complejidad temporal del problema es de orden N^2 con unos tiempos de respuesta superiores a un segundo para puzzles de dimensiones de 60x60 en

adelante. Esto es debido principalmente a que los algoritmos de comparación de imágenes utilizados por el momento realizan dicha comparación evaluando el código de color RGB píxel a píxel.

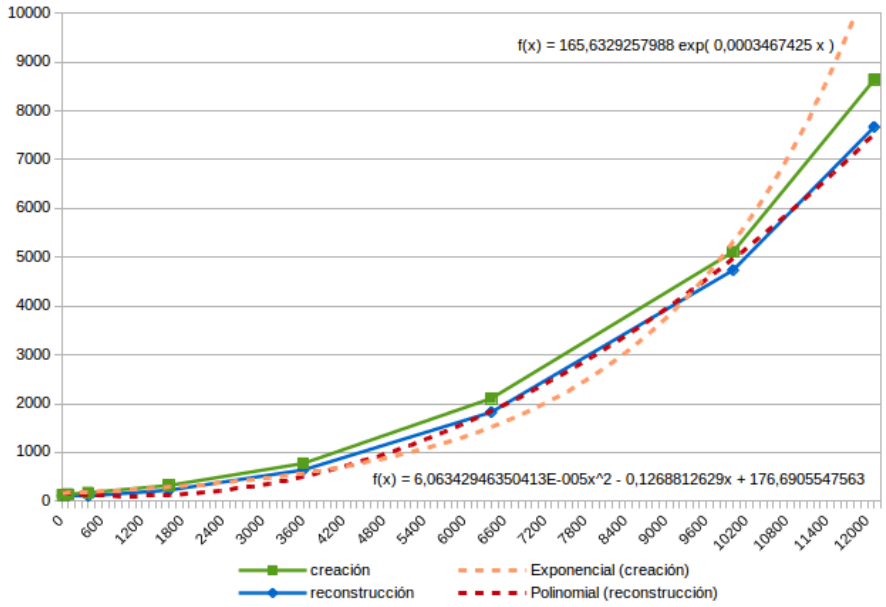

Figura 1: *Complejidad del problema*

3 - Diseño e implementación

En la *Figura 2* se muestra una representación del Diagrama de clases de Diseño elaborado previamente a la implementación del problema. A continuación se explican detalladamente cada una de las decisiones de diseño y la implementación realizada de cada una de las clases.

3.1 - Puzzle

Como se puede observar en la la *Figura 1*, el artefacto Puzzle de queda definido por dos arrays, uno bidimensional de dimensiones N columnas por M filas compuesto por elementos de tipo Pieza y un array unidimensional que almacenará la posición i,j de la pieza vacía. A continuación se explica la funcionalidad e implementación de aquellos métodos más relevantes de la clase Puzzle.

3.1.1 - creacion()

Este método es el encargado de crear un elemento puzzle original de dimensiones (cols x rows)-1 piezas a partir de un fichero path con extensión .png.

Como se observa en el *Listado 1 y Listado 2*, este método realiza una llamada al constructor Puzzle que será en encargado de llevar a cabo la construcción del artefacto puzzle con los parámetros proporcionados en la llamada. En el código del constructor se puede ver como éste realiza una llamada al método *splitImages()* que será el encargado del almacenamiento, manipulación y generación de imágenes de cada una de las piezas que conformarán el puzzle.

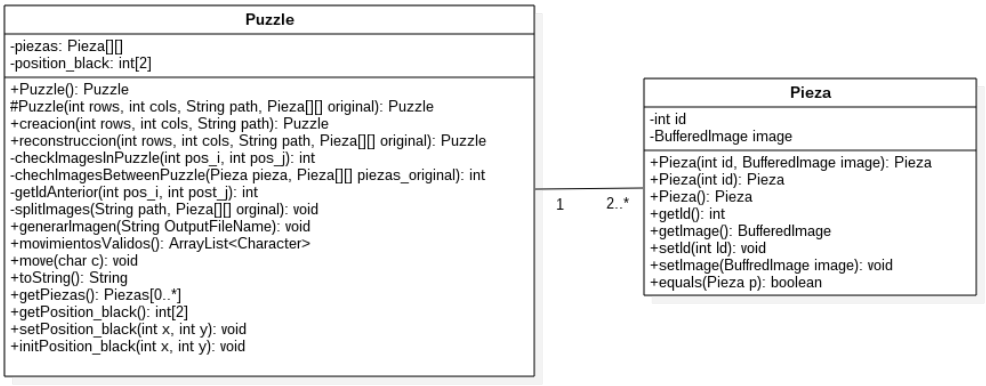

Figura 2: *Diagrama de clases de Diseño*

```
public Puzzle creacion(int rows, int cols, String path) throws IOException{
        long ini = obtenerTiempo('M');
        Puzzle original = new Puzzle(rows, cols, path, null);
        long fin = obtenerTiempo('M');
        System.out.println("Tiempo total creación del puzzle: "+(fin-ini)+"ms");
        return original;
}
```

Listado 1: *implementación del método creacion()*

```
protected Puzzle(int rows, int cols, String path, Pieza[][] original) throws IOException{
        this.piezas = new Pieza[rows][cols];
        this.position_black=new int[2];
        splitImages(path, original);
}
```

Listado 2: *Constructor Puzzle()*

3.1.2 - reconstruccion()

Este método es el encargado reconstruir un elemento Puzzle de dimensiones NxM-1 piezas a partir de un fichero con extensión .png correspondiente a la imagen de un artefacto Puzzle modificado.

Al igual que ocurre con la creación, este método realiza una llamada al constructor Puzzle con la única diferencia de que el último parámetro pasado al constructor no es nulo, si no el array de objetos Pieza del Puzzle original. Los distintos comportamientos del método *splitImages()* según el valor de este parámetro se detallan a continuación en la sección 3.1.3.

```
public Puzzle reconstruccion(int rows, int cols, String path, Pieza[][] original){
        long ini = obtenerTiempo('M');
        Puzzle modificado = new Puzzle(rows, cols, path, original);
        long fin = obtenerTiempo('M');
        System.out.println("Tiempo total reconstruccion del puzzle: "+(fin-ini)+"ms");
        return modificado;
}
```

Listado 3: *implementación del método reconstruccion()*

3.1.3 - splitImage()

Este método es el encargado de recortar la imagen “path” pasada como parámetro en NxM imágenes más pequeñas, según el número de columnas y filas especificadas del array de objetos

Pieza “original”. Para llevar a cabo la implementación se ha hecho uso de los objetos *BufferedImage* y *Graphic2D* de las librerías *java.awt* y *javax.imageio* para el almacenamiento, manipulación y generación de imágenes.

Además, este método se encarga de la comparación de imágenes en el puzzle original y entre el original y el modificado, con ayuda de los métodos *checkImagesInPuzzle(i, j)* y *checkImagesBetweenPuzzle(Pieza pieza, Pieza[][] piezas_original)*, según sea el valor del parámetro “original” pasado como argumento. Se explican a continuación los dos comportamientos que se han implementado:

1. Para la fase de creación del artefacto Puzzle original dicho método recibe como parámetro “original” = “null” y, por tanto, la comparación de imágenes para la asignación de id’s se realiza entre las imágenes que van conformando al puzzle original en cada iteración, realizando una invocación al método *checkImagesInPuzzle(i, j)*. En el caso en el que la nueva imagen coincida con una ya existente en el puzzle se le asignará el mismo id que la pieza existente y uno nuevo en caso contrario.

2. Para la fase de creación del artefacto Puzzle modificado a partir de la imagen resultante tras la reconstrucción de un puzzle, dicho método recibe como parámetro el array de objetos Piezas del Puzzle original. Puesto que todas y cada una de las imágenes de las piezas del puzzle original y modificado deben coincidir, por cada una de las iteraciones se realiza una invocación al método *checkImagesBetweenPuzzle(getPiezas()[i][j])* que compara si la pieza que va a incorporarse al puzzle modificado coincide con alguna del puzzle original. En caso de coincidencia se le asignará a dicha pieza el id de la pieza del puzzle original con la que coincide la imagen. En caso contrario se aborta la ejecución del método *splitImages()* ya que se tratan de puzzles diferentes y nunca se alcanzaría la solución.

```
public void splittImage(String path,Pieza[][] piezas_original) throws IOException{
/********Código anterior*********/
if(piezas_original==null){
        if(i==0 && j==0){
                gr.setColor(Color.BLACK);
                getPiezas()[i][j].setId(0);
        }
        else{
                gr.drawImage(image, 0, 0, cellWidth, cellHeight, cellWidth * j, cellHeight *
i, cellWidth * j + cellWidth, cellHeight * i + cellHeight, null);
                id=checkImagenesInPuzzle(i, j);
                if(id==-1) getPiezas()[i][j].setId(getIdAnterior(i, j)+1);
                        else getPiezas()[i][j].setId(id);
                }
        }
        else{
                gr.drawImage(image, 0, 0, cellWidth, cellHeight, cellWidth * j, cellHeight *
i, cellWidth * j + cellWidth, cellHeight * i + cellHeight, null);
                id=checkImagesBetweenPuzzle(getPiezas()[i][j], piezas_original);
if(id!=-1){
                        getPiezas()[i][j].setId(id);
                        if(id==0) initPosition_black(i, j);
                }
                else{
                        System.out.println("NO SOLUTION");
                        getPiezas()[0][0].setId(-1);
        }
}
/********Código posterior********/
}
```

Listado 4: *implementación del método splitImage()*

3.1.4 - generarImagen()

Este método es el encargado de generar un fichero gráfico con extensión .png a partir de las imágenes de cada una de las piezas que conforman el artefacto Puzzle tomando como parámetro el nombre de fichero de salida.

De un modo similar que en la implementación del método *splitImage()*, para la generación de imágenes se ha hecho uso de los objetos *BufferedImage* e *ImageIO* de las librerías *java.awt* y *javax.imageio* para la manipulación y almacenamiento de imágenes.

```
       public void generarImagen(String OutputFileName) throws IOException{
               int cellWidth = this.getPiezas()[0][0].getImage().getWidth();
               int cellHeight = this.getPiezas()[0][0].getImage().getHeight();
               int type = this.getPiezas()[0][0].getImage().getType();
               BufferedImage                resultImg                =                new
BufferedImage(cellWidth*this.getPiezas()[0].length,      cellHeight*this.getPiezas().length,
type);
               for (int i = 0; i < this.getPiezas().length; i++) {
                   for (int j = 0; j < this.getPiezas()[i].length; j++) {

resultImg.createGraphics().drawImage(this.getPiezas()[i][j].getImage(),   cellWidth   *   j,
cellHeight * i, null);
                   }
               }
               System.out.println("Image "+OutputFileName+".png concatenated.....");
               ImageIO.write(resultImg, "png", new File(OutputFileName+".png"));
       }
```

Listado 5: *Implementación del método generarImagen()*

3.1.5 - movimientosValidos()

Este método es el encargado de generar una lista de movimientos válidos según {“U”, “D”, “L”, “R”} (Up, Down, Left, Right) en función de la posición que ocupe la pieza vacía del artefacto puzzle.

Estos movimientos válidos o permitidos vienen determinados por las limitaciones físicas del puzzle, esto es, el contorno del mismo. Como se observa en la implementación, todas aquellas posiciones de la pieza vacía que no coincida con alguno de los laterales o esquinas no limitarán los posibles movimientos sobre esta. Sin embargo, si dicha pieza se encuentra en una esquina se limitarán al menos un movimiento vertical y otro horizontal mientras que únicamente se limitará un movimiento, bien horizontal o vertical, si la posición coincide con alguno de los laterales del puzzle.

```
public ArrayList<Character> movimientosValidos(){
       int rows=getPiezas().length;
       int cols=getPiezas()[0].length;
       ArrayList<Character> list=new ArrayList<Character>();
       if(getPosition_black()[0]!=0) list.add('U');
       if(getPosition_black()[0]!=rows-1) list.add('D');
       if(getPosition_black()[1]!=0) list.add('L');
       if(getPosition_black()[1]!=cols-1) list.add('R');
return list}
```

Listado 6: *Implementación del método movimientosValidos()*

3.1.6 - checkImagesInPuzzle()

Este método compara si la imagen correspondiente a las pieza[pos_i][pos_j] del Puzzle coincide con alguna de las imágenes de las piezas existentes en el mismo. En caso de coincidencia devuelve el id de la pieza coincidente y en el caso en el imágen no exista previamente en el puzzle

devuelve un nuevo id para su asignación.En caso de coincidencia se le asignará a dicha pieza el id de la pieza del puzzle original con la que coincide la imagen

```
        public int checkImagenesInPuzzle(int pos_i, int pos_j){
                int id=-1;
                int rows=getPiezas().length;
                int cols=getPiezas()[0].length;
                for(int i=0; i<rows; i++){
                        for(int j=0; j<cols; j++){
                                if(i==pos_i && j==pos_j) return id;
                                if(getPiezas()[i][j].equals(getPiezas()[pos_i][pos_j])){
                                        id=getPiezas()[i][j].getId();
                                        return id;
                                }
                        }
                }
                return id;
        }
```

Listado 7: *Implementación del método checkImagesInPuzzle()*

3.1.7 - checkImagesBetweenPuzzle()

Este método compara las imágenes entre los objetos Pieza que conforman los artefactos Puzzle original y modificado garantizando la correspondencia entre estos. En caso de coincidencia se le asignará a dicha pieza el id de la pieza del puzzle original con la que coincide la imagen. En caso contrario se devuelve un id con valor -1 en concepto de error ya que si no existe coincidencia entre imágenes ambos puzzles serían diferentes y nunca se alcanzaría la solución.

```
public int checkImagesBetweenPuzzle(Pieza pieza, Pieza[][] piezas_original){
        int id = -1;
        int rows = piezas_original.length;
        int cols = piezas_original[0].length;
        for(int i=0; i<rows && id==-1; i++){
                for(int j=0; j<cols && id==-1; j++){
                        if(pieza.equals(piezas_original[i][j])){
                                id = piezas_original[i][j].getId();
                        }
                }
        }
        return id;
}
```

Listado 8: *Implementación del método checkImagesBetweenPuzzle()*

3.2. - Pieza

Como se puede observar en la la *Figura 1*, los objetos Pieza están compuestos por una imagen y un identificador de la misma que nos permitirá gestionar de manera más eficiente la correspondencia imagen-pieza del puzzle una vez este sea manipulado mediante la realización de movimientos. A continuación se explica la funcionalidad e implementación de aquellos métodos más relevantes de la clase Puzzle.

3.2.1 - equals

Como ya se comentó en la sección 3.1, este método se encarga de la comparación de imágenes entre piezas devolviendo un booleano igual a true si existe coincidencia entre las imágenes de los objetos pieza invocante y pasado como parámetro y false en casa contrario,

Al margen de la finalidad del método, es importante destacar cómo se realiza la comparación de imágenes. Si se observa el código, por el momento y aunque se trata de una técnica poco eficiente, en esta primera versión la comparación se realiza píxel a píxel entre ambas imágenes haciendo uso del método *getRGB(i,j)* de la clase *BufferedImage* del paquete java.awt.

```
public boolean equals(Pieza p){
        if(getImage().getWidth()!=p.getImage().getWidth()||getImage().getHeight()!=p.getImag
e().getHeight())       return false;
        int w = getImage().getWidth();
        int h = getImage().getHeight();
        for(int i=0; i<h; i++) {
                for(int j=0; j<w; j++) {
                        if(getImage().getRGB(j,i) != p.getImage().getRGB(j,i))
                                return false;
                }
        }
        return true;
}
```

Listado 8: *Implementación del método equals()*

HITO 2

---

1 - Introducción

Una vez definido e implementado el artefacto Puzzle junto con sus operaciones básicas en el Hito 1, se procede a la definición y codificación del Problema, del Espacio de Estados y de la Frontera necesarios para la simulación del árbol de búsqueda, exploración y obtención de soluciones sobre estados del artefacto Puzzle.

Además, se llevará a cabo un análisis de los tiempos de ejecución para diferentes estructuras de datos, según tiempo mínimo, máximo y medio en la inserción de elementos sobre cada una de ellas, de modo que nos permita conocer qué estructura de datos presenta mejor rendimiento para ser utilizada como Frontera.

2 - Diseño e implementación

En el siguiente diagrama UML de la *Figura 3* se muestra una representación del Diagrama de clases de Diseño elaborado previamente a la implementación de los componentes que intervienen en la definición del problema e implementación de la frontera.

A continuación se explican detalladamente cada una de las decisiones de diseño y la implementación realizada de cada una de las clases.

2.1 - Problema

Como se puede observar en la *figura 3*, la clase Problema queda definida por un Estado, que representa el estado resultante de la reconstrucción de una imagen modificada, y un Espacio de Estados que, a modo de resumen, es una clase encargada de generar los sucesores a partir de una acción y estado cualquiera, y de comprobar si un estado es válido y es objetivo, es decir, que es posible alcanzar una solución a partir de este y comprobar si dicho estado coincide con el estado meta.

A continuación se explica la funcionalidad e implementación de aquellos métodos más relevantes de la clase Problema.

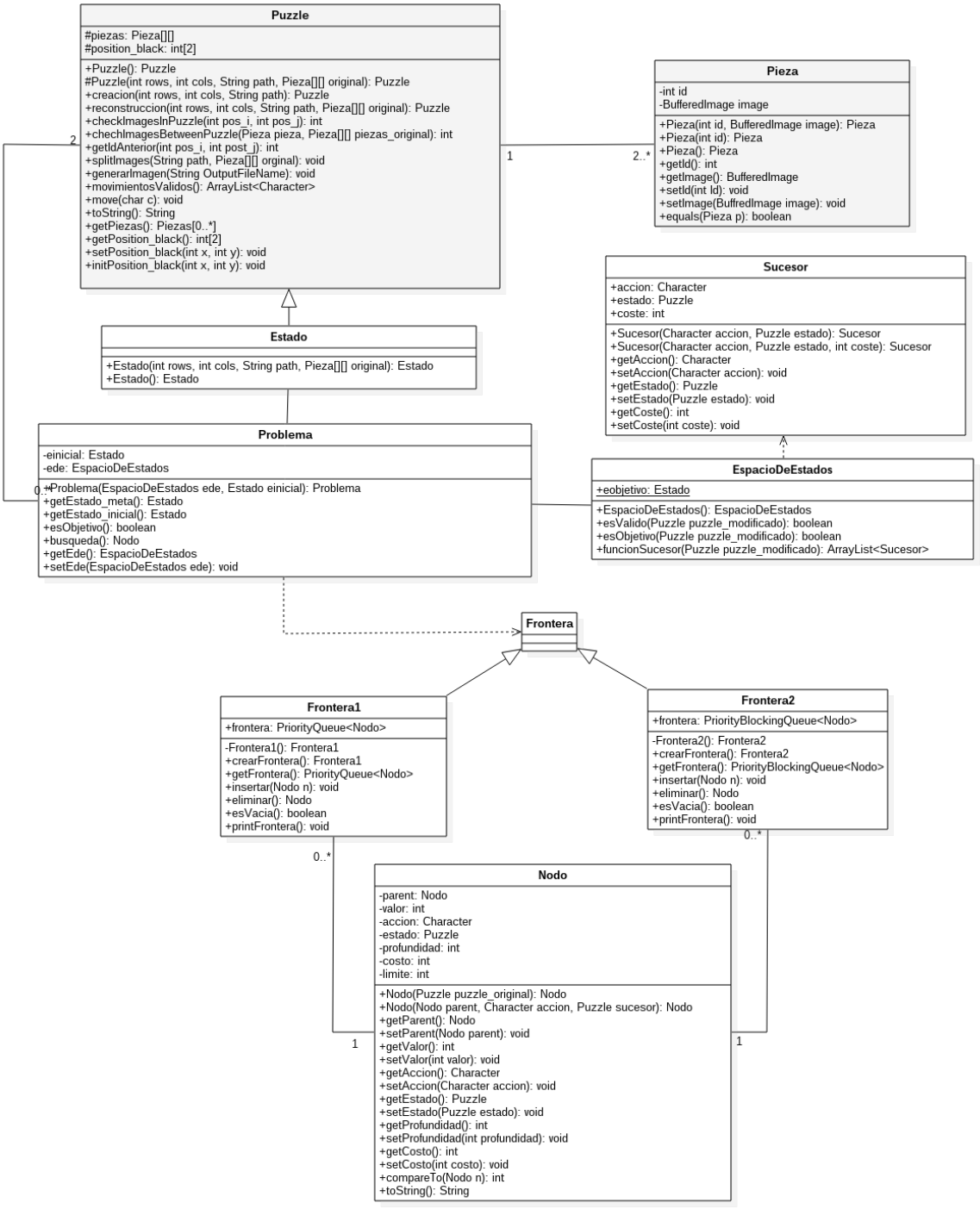

Figura 3 *- Diagrama de clases de Diseño*

2.1.1 - Problema()

Este método (Constructor) crea una instancia de la clase Problema inicializando los objetos de tipo Estado *einicial* y EspacioDeEstados *ede* al de los objetos del mismo tipo pasados como parámetros al constructor. Es decir, para la creación de un Problema será necesario definir un Estado inicial desde el que comience la búsqueda de soluciones y un Espacio de Estados para explorar nuevos estados en búsqueda del estado final deseable.

```
public Problema(EspacioDeEstados ede, Estado einicial) throws IOException{
	this.ede=ede;
	this.einicial=einicial;
}
```

Listado 9: *constructor Problema()*

2.1.2 - esObjetivo()

Este método que recibe un Estado cualquiera y devuelve un booleano en función de si dicho estado coincide con el estado objetivo. Para llevar a cabo esta comparación, el método *esObjetivo()* se apoya en el método de igual signatura de la clase *EspacioDeEstados*.

```
public boolean esObjetivo(Estado Estado_modificado){
	return ede.esObjetivo(Estado_modificado);
}
```

Listado 10*: implementación del método esObjetivo()*

2.1.3 - busqueda()

Este método es el encargado de lanzar el algoritmo de búsqueda. Para ello es necesario generar una instancia de la clase *Frontera* que almacene aquellos nodos del árbol de búsqueda que han sido expandidos y no explorados en las diferentes iteraciones. Esta frontera inicialmente almacenará el Nodo correspondiente al Estado inicial del problema, es decir, einicial, y comenzará la exploración de nodos.

1. Si la Frontera no está vacía, se selecciona y se extrae el Nodo con menor valor existente en la frontera.
2. Se comprueba si el estado del Nodo seleccionado es un estado objetivo. En caso afirmativo terminaría la ejecución. En caso contrario, se expanden los sucesores del Nodo seleccionado mediante la llamada al método *funcionSucesor(Nodo n_actual.getEstado)* del EspacioDeEstados.
3. Si se han generado nuevos sucesores a partir del Nodo seleccionado, a continuación se generan los nuevos Nodos a incorporar en la Frontera mediante la llamada al constructor *Nodo(Nodo parent, Character accion, Puzzle sucesor)* de la *clase Nodo*.
4. Los Nodos sucesores se insertan en la frontera y se repite el proceso.

```
public void busqueda() throws IOException{
	try {
		Nodo n_actual = null;
		Frontera1 frontera=Frontera1.crearFrontera();
		Nodo n_inicial=new Nodo(getEinicial());
		frontera.insertar(n_inicial);
		System.out.println("Generando arbol...");
		while(true){
			n_actual = frontera.eliminar();
			ArrayList<Sucesor> sucesores =
getEde().funcionSucesor(n_actual.getEstado());
			for(int i=0; i<sucesores.size(); i++){
				frontera.insertar(new Nodo(n_actual,
sucesores.get(i).getAccion(), sucesores.get(i).getEstado()));
			}
		}
	} catch (OutOfMemoryError e) {
		imprimeTiempos(Frontera1.tinsert, new FileWriter("tiempos_insercion1.txt"));
	}
}
```

***Listado 11**: implementación del método buscarSolucion()*

Sin embargo, por ahora la búsqueda de soluciones al problema no es tan importante como la selección de una estructura de datos adecuada para la implementación de la Frontera. Por este motivo, la implementación de método *búsqueda()* incluye la ejecución de un bucle infinito en el que se generan nuevos Sucesores, se expanden los Nodos y se incluyen en la Frontera sin comprobar si el estado de un nodo es o no objetivo. Esto supondrá la ocupación de la memoria Heap de la Máquina Virtual de Java generando una excepción de tipo OutOfMemoryError. Una vez capturada la excepción, se escriben en un fichero los tiempos de inserción de cada uno de los Nodos en las Estructuras de Datos utilizadas para la implementación de la Frontera.

2.2 - Estado

Como se puede observar en la *Figura 3,* un Estado es un representación del artefacto Puzzle llevada a cabo mediante una herencia, de modo que un Estado hereda todos los atributos y funcionalidades de su objeto padre, en este caso, un objeto Puzzle. De esta manera se consigue abstraer la búsqueda de soluciones de la naturaleza del problema pudiendo reutilizar gran parte de la implementación para la resolución de otro tipo de problemas.

En el *Listado 13* se incluye la implementación de los dos únicos métodos constructores que se añaden en la clase Estado, al margen del resto de funcionalidad heredada de la clase Puzzle.

```
public Estado(){}

public Estado(int rows, int cols, String path, Pieza[][] original) throws IOException{
       super(rows, cols, path, original);
}
```

***Listado 12**: Constructores de la clase Estado*

2.3 - Espacio de Estados

Como se puede observar en la *figura 3,* la clase *EspacioDeEstados* queda definida por únicamente un Estado *eobjetivo*, que representa el estado meta y que es el resultado del proceso de creación de un artefacto Puzzle a partir de una imagen con extensión *.png. Además es la encargada de generar los sucesores a partir de un estado cualquiera y de comprobar si un estado es válido y es objetivo.

A continuación se explica la funcionalidad e implementación de aquellos métodos más relevantes de la clase EspacioDeEstados.

2.3.1 - esVálido()

Este método comprueba si a partir de un estado cualquiera es posible alcanzar el *estado objetivo*, es decir, es posible obtener una solución al problema. Gran parte del trabajo necesario para tomar esta decisión es realizado en la fase de reconstrucción del artefacto Puzzle, en la cual se asigna un id = -1 en la posición [col=0, row=0] si alguna de las subimagenes no coincide con alguna de las subimagenes del puzzle original. De este modo, dado un estado, consultando el id la posición [col=0, row=0] es posible conocer si el estado es o no válido.

```
public boolean esValido(Estado Estado_modificado){
```

```
        boolean valido = true;
        if(Estado_modificado.getPiezas()[0][0].getId()==-1){
                valido=false;
        }
        return valido;
}
```

***Listado 13**: implementación del método esValido()*

2.3.2 - esObjetivo()

Este método comprueba si un estado cualquiera pasado como parámetro es igual al del estado meta *eobjetivo*. Para ello, el método comprueba posición a posición el id de las piezas (Estado) que componen el artefacto Puzzle. En el momento en el que el id de *estado_modificado[row][col]* no coincida con el id de *eobjetivo[row][col]* finaliza la ejecución del bucle y el método devuelve false. En el caso en el que todos los id's coincidan se habrá alcanzado un estado igual a *eobjetivo* y el método devolverá true.

```
public boolean esObjetivo(Estado Estado_modificado){
        int id = 0;
        boolean esObjetivo = true;
        for(int i = 0; i< Estado_modificado.getPiezas().length && esObjetivo; i++)
                for(int j=0;j<Estado_modificado.getPiezas()[i].length && esObjetivo; j++)
if(Estado_modificado.getPiezas()[i][j].getId()!=eobjetivo.getPiezas()[i][j].getId())
esObjetivo=false;
        return esObjetivo;
}
```

***Listado 14**: implementación del método esObjetivo()*

2.3.3 - funcionSucesor()

Este método genera todos los sucesores a partir de un estado pasado como parámetro en base a los movimientos válidos que puedan ejecutarse sobre dicho estado del puzzle. Para ello, este método se apoya en las funciones *movimientosValidos()* y *move()* de la clase *Puzzle,* que generan una lista de movimientos posibles y realizan dichos movimientos sobre un puzzle, respectivamente. Este método devuelve un ArrayList de objetos *Sucesor* con la siguiente estructura *['Acción', 'Estado resultante', 'Coste Acción'] (Ver implementación clase Sucesor.java)*.

```
public ArrayList<Sucesor> funcionSucesor(Estado Estado_modificado){
        ArrayList<Sucesor> sucesores = new ArrayList<Sucesor>();
        ArrayList<Character> movimientos = Estado_modificado.movimientosValidos();
        for(int i = 0; i < movimientos.size(); i++){
                Estado aux = new Estado();
                Pieza[][] piezas_sucesor = new
Pieza[Estado_modificado.getPiezas().length][Estado_modificado.getPiezas()[0].length];
                for(int j=0; j<Estado_modificado.getPiezas().length;j++)
                  for(int k=0; k<Estado_modificado.getPiezas()[0].length;k++)
                        piezas_sucesor[j][k]=Estado_modificado.getPiezas()[j][k];
                        aux.setPiezas(piezas_sucesor);
                        aux.initPosition_black(Estado_modificado.getPosition_black()[0],
Estado_modificado.getPosition_black()[1]);
                        aux.mover(movimientos.get(i));
                        Sucesor sucesor = new Sucesor(movimientos.get(i), aux);
                        sucesores.add(sucesor);
        }
        return sucesores;
}
```

***Listado 15**: implementación del método funcionSucesor()*

2.4 - Frontera

Como se puede observar en la *Figura 3,* para la implementación de la Frontera se han utilizado dos Estructuras de Datos dinámicas que proporciona *Java,* en particular una *PriorityQueue<Nodo>* y una *PriorityBlockingQueue,* con el objetivo de analizar los tiempos de inserción de Nodos en la frontera en diferentes implementaciones y poder justificar así la utilización de una u otra en particular. El análisis de rendimiento y técnicas de ordenación de Nodos en la Frontera se detalla en mayor profundidad en la *sección 3*.

Aunque las implementaciones de las clases *Frontera1 y Frontera2* son diferentes debido a la utilización de varios tipos de estructuras de datos, sí que mantienen la misma interfaz en las funcionalidades relacionadas con la creación de la Frontera, inserción y eliminación de nodos.

2.4.1 - crearFrontera()

Este método realiza una llamada al Construtor de la clase y devuelve una instancia creada de Frontera1 o Frontera2 (*PriorityQueue<Nodo> o PriorityBlockingQueue<Nodo>).*

```
/*Frontera1*/
public static Frontera1 crearFrontera() {
        return new Frontera1();
}
/*Frontera2*/
public static Frontera1 crearFrontera() {
        return new Frontera2();
}
```

Listado 16: *implementación del método crearFrontera()*

2.4.2 - insertar()

Método que se encarga de insertar en la Frontera un objeto Nodo. Para cualquier estructura de datos utilizada el Nodo de menor *valor* se insertará en la primera posición de la Frontera. Para ello la clase *Nodo* implementa la interfaz *comparable* de java, cuya implementación compara y ordena los elementos dentro de la estructura de datos de manera creciente según el valor del Nodo.

```
/*Frontera1*/
public void insertar(Nodo n) {
        tinsert1=Stat.obtenerTiempo('N');
        this.frontera.add(n);
        tinsert2=Stat.obtenerTiempo('N');
        tinsert.add((tinsert2-tinsert1));
}
/*Frontera2*/
public void insertar(Nodo n){
       tinsert1=Stat.obtenerTiempo('N');
        this.frontera.add(n);
        tinsert2=Stat.obtenerTiempo('N');
        tinsert.add((tinsert2-tinsert1));
}
```

Listado 17*: implementación del método eliminar()*

2.4.3 - eliminar()

Método que se encarga de devolver y eliminar el Nodo con el menor *valor* de la Frontera. Puesto que la implementación de la interfaz comparable garantiza la ordenación de los elementos de la Frontera en orden creciente según el valor del Nodo, el nodo a eliminar será siempre el que ocupe la primera posición de la estructura de datos.

```
/*Frontera1*/
public Nodo eliminar() {
        return this.frontera.remove();
}

/*Frontera2*/
Public Nodo eliminar(){
       Nodo n = this.frontera.get(0);
        this.frontera.remove(0);
        return n;
}
```

Listado 18: *Implementación del método insertar()*

2.5 - Nodo

Como se puede observar en la *Figura 3,* la clase Nodo queda definida por un total de seis atributos y dos constructores. Entre estos atributos, es posible establecer la siguiente clasificación atendiendo a la información que manejan:

- Información del Padre: El atributo *parent* es el encargado de enlazar cada uno de los Nodos generados con su Nodo padre. Esta relación permitirá obtener una solución al problema una vez encontrado un nodo cuyo estado coincida con el estado objetivo.
- Información del Dominio:
    - Estado: Almacena la representación actual del Puzzle
    - Acción: Movimiento que ha dado lugar al nuevo estado del puzzle con respecto al padre
    - Coste: Valor que supone alcanzar el nodo actual desde el nodo inicial. Si el estado del nodo es objetivo coincide con el coste de la solución.
    - Profundidad: Nivel que ocupa el nodo en el Árbol de Búsqueda
    - Valor: Clave numérica mediante la cual se ordenarán los nodos de la Frontera

A continuación se explica la funcionalidad e implementación de aquellos métodos más relevantes de la clase Nodo.

2.5.1 - Nodo()

Existen dos constructores que intervienen en la generación de objetos Nodo en esta clase. El constructor de *Listado 19* es el encargado de instanciar el primer Nodo de la Frontera, es decir, el Nodo padre del Árbol de Búsqueda. Éste presenta una serie de particularidades frente al resto de Nodos de árbol debido a que únicamente posee un *estado* que corresponde al estado inicial del puzzle y un *valor* aleatorio entre 1 y *limite*, mientras que el resto de atributos toman valores nulos.

```
public Nodo(Puzzle puzzle_inicial){
        this.profundidad = 0;
        this.parent = null;
        this.accion = null;
        this.estado = puzzle_inicial;
        this.valor = (int)(limite*Math.random()+1);
        this.costo = 0;
}
```

Listado 19: *constructor Nodo()*

Por otro lado, el constructor del *Listado 20* se encarga de instanciar cualquier Nodo de la frontera, enlazando su atributo Nodo con el Nodo *parent* e inicializando el atributo *acción* y *estado* a

los pasados por parámetro al constructor. Además, los atributos *costo* y profundidad se igualan a los valores de coste y profundidad del padre incrementados en una unidad, ya que la profundidad de un nodo es una unidad mayor que la de su padre y se asume un coste unitario por acción. Al igual en el constructor anterior, valor toma un número entero aleatorio entre 1 y *limite = 50*.

```
public Nodo(Nodo parent, Character accion, Puzzle sucesor){
               this.parent = parent;
               this.profundidad = this.parent.getProfundidad()+1;
               this.accion = accion;
               this.estado = sucesor;
               this.valor = (int)(limite*Math.random()+1); //recorrido aleatorio
               this.costo = this.parent.getCosto()+1;
       }
```

Listado 20: *constructor Nodo()*

2.5.2 - compareTo()

Este método permite ordenar objetos Nodo dentro de la frontera atendiendo al valor de Nodo. Para una ordenación creciente según el valor, se ha implementado del siguiente modo el método *compareTo()* de la interfaz *Comparable<Nodo>:*

```
public int compareTo(Nodo n){
       if(getValor() < n.getValor()) return -1;
       else if(getValor() > n.getValor()) return 1;
       return 0;
}
```

Listado 21: *implementación del método compareTo()*

3 - Frontera: Estructuras de Datos y Análisis de tiempos

Debido a la necesidad de insertar y eliminar nodos en la frontera de nuestro árbol de búsqueda, la estructura de datos elegida debía ser una estructura dinámica que hiciera posible un tamaño redimensionable. Como bien se ha mencionado en el apartado de introducción hemos llevado a cabo un análisis de los tiempos con diferentes estructuras de datos para determinar cuál presenta un mejor rendimiento, utilizando y analizando PriorityQueue y PriorityBlockingQueue.

Fueron tanteadas otras opciones como HashMap o TreeMap, pero debido a su naturaleza de relación clave-valor (definiendo cada clave como el valor del nodo), se hacía inviable una implementación de la frontera usando cualquiera de estas estructuras de datos, debido a la posibilidad de que varios nodos compartan un mismo valor y por tanto, existieran colisiones entre claves.

La clase Nodo, usada para crear nodos que almacenará nuestra frontera, implementa la Interfaz Comparable, lo que le permite mediante el método compareTo() comparar nodos dos a dos atendiendo a su valor, y cuyo resultado es usado por las distintas estructuras de datos usadas como frontera para ordenar sus nodos, en el caso que nos atiende, en orden creciente.

Dado que los dos tipos de estructuras de datos usadas tienen implementaciones dinámicas, el número máximo de nodos que pueden ser almacenados en cualquiera de ellas no viene limitado

por el tipo de estructura en sí, sino por la cantidad de memoria Heap disponible en la Máquina Virtual de Java.

3.1 – PriorityQueue

La estructura de datos PriorityQueue admite la inserción de elementos en un orden arbitrario. Sin embargo, el “removal” de elementos se lleva a cabo siguiendo el orden de prioridad establecido, garantizando así que el método remove() recoja y elimine de la frontera el Nodo con menor valor, que estará en la cabeza de la cola .

Analizando los tiempos de inserción mínimos, máximo y medio de nodos en este tipo de estructura observamos los siguientes resultados:

- Tiempo mínimo de inserción =18 ns
- Tiempo máximo de inserción = 47214 ns
- Tiempo medio de inserción = 58.713 ns

Como se observa en la *Figura 4* los tiempos de inserción en la Frontera, implementada mediante una PriorityQueue, se mantienen de manera general por debajo de los 500ns presentando un tiempo medio de inserción de 58ns y una tendencia lineal prácticamente despreciable al número de Nodos almacenados en la Frontera.

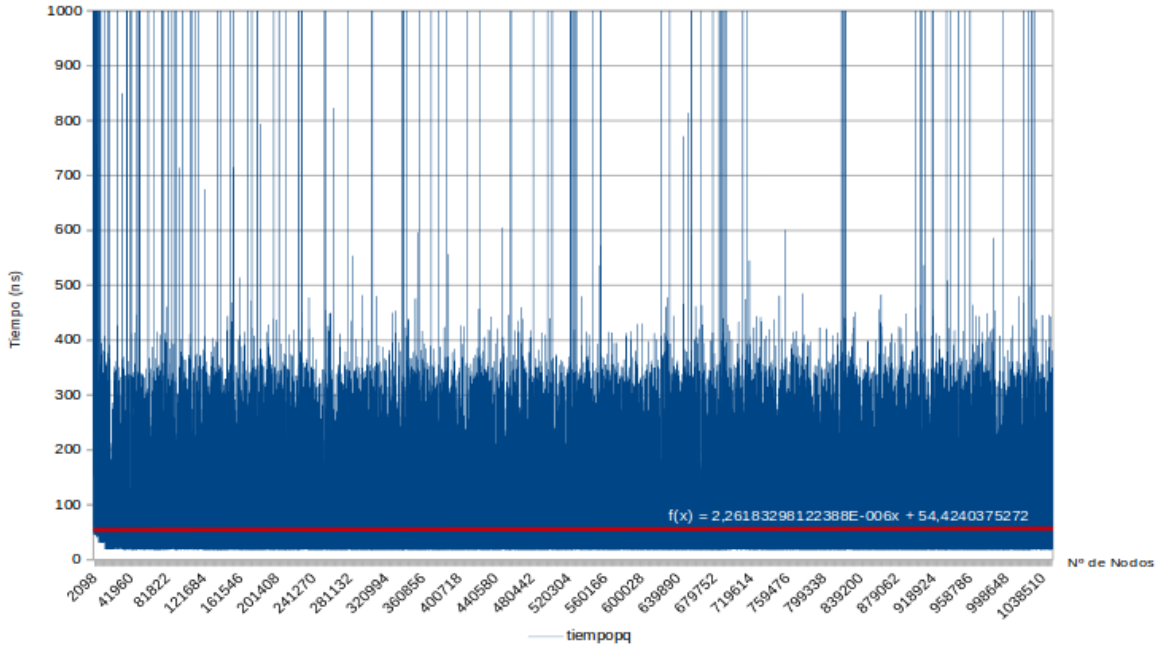

Figura 4: *Gráfico Tiempo-Inserción Nodo PriorityQueue*

3.2 – PriorityBlockingQueue

La estructura de datos PriorityBlockingQueue, al igual que en PriorityQueue, admite la inserción de elementos en un orden arbitrario. Del mismo modo, el “removal” se lleva a cabo siguiendo el orden de prioridad establecido, garantizando así que el método remove() devuelva y elimine de la frontera el Nodo con menor valor, que ocupará la cabeza del a cola.

Para este planteamiento los tiempos obtenidos son los siguientes:

- Tiempo mínimo de inserción= 29 ns
- Tiempo máximo de inserción= 88006 ns
- Tiempo medio de inserción= 76.624 ns

Como se observa en la *Figura 5* los tiempos de inserción en la Frontera, implementada mediante una PriorityBlockingQueue, se mantienen de manera general por debajo de los 500ns presentando un tiempo medio de inserción de 76ns y una tendencia lineal con una pendiente superior a la que presenta PriorityQueue.

3.3 – Conclusión

Tomando como referencia los tiempos medios obtenidos para cada estructura de datos en sus diferentes usos y los gráficos de la *Figura 3* y *Figura 4* podemos concluir que los tiempos medios de inserción de nodos, aunque son muy similares, son superiores en la estructura de datos del tipo PriorityBlockingQueue. Además, la tendencia de este tipo de estructura de datos es mayor que la tendencia obtenida para el tipo PriorityQueue, por lo que presenta un peor rendimiento para tamaños de problema mayores. Por todo ello, no resulta difícil determinar que la estructura de datos idónea para usar como Frontera en nuestro árbol de búsqueda es PriorityQueue.

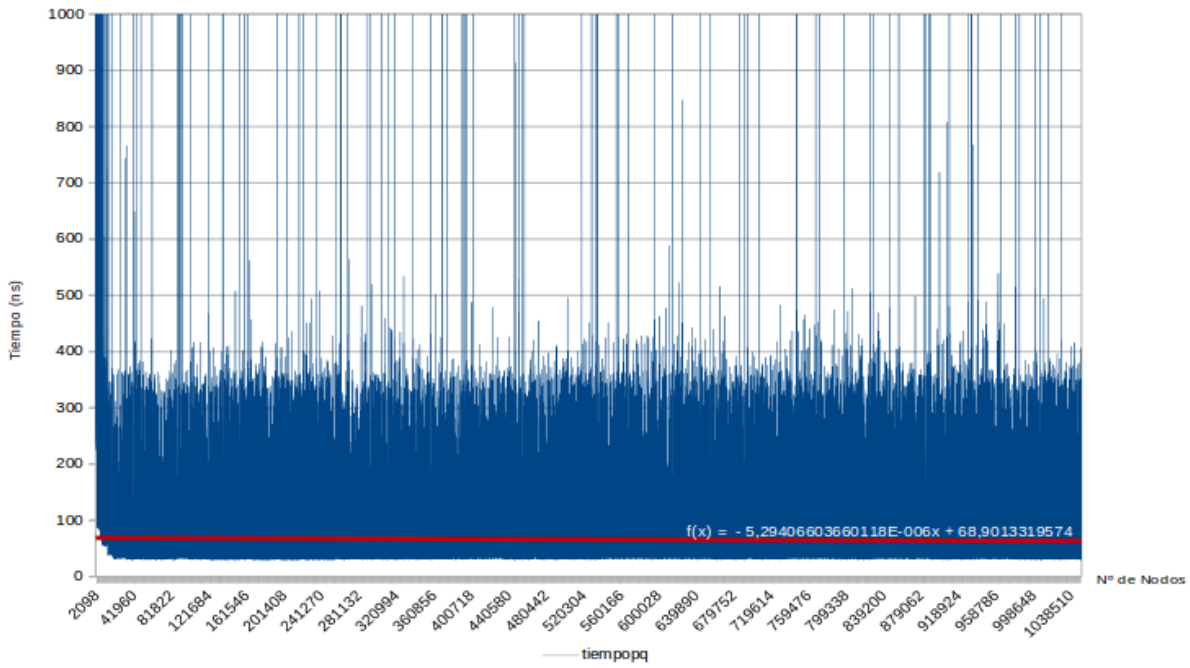

Figura 5: *Gráfico Tiempo-Inserción Nodo PriorityBlockingQueue*

HITO 3

---

1 - Introducción

Una vez implementados todos los elementos necesarios para la definición de un Problema en los anteriores hitos, esta tarea se centra en la realización de un algoritmo básico de búsqueda que permita implementar las siguientes estrategias:

1. Anchura
2. Profundidad Simple, Acotada y Acotada Iterativa
3. Costo Uniforme

El objetivo es diseñar un algoritmo de búsqueda que pueda ser reutilizado para la implementación de las diferentes estrategias mencionadas de modo que, haciendo llamadas a un único de algoritmo, podamos generar diferentes árboles de búsqueda mediante la utilización de diferentes estrategias.

Para alcanzar el objetivo de este hito se cuenta con la siguiente información de partida:

1. Clase Problema: que deberá instanciar un objeto de la clase *EspacioDeEstados* y un estado inicial como representación del puzzle a partir de la reconstrucción de un artefacto Puzzle.
2. Clase *EspacioDeEstados*: que deberá instanciar un estado del artefacto Puzzle almacenando una configuración concreta de todas la piezas del puzzle en su estado deseable u objetivo.
3. Clase Nodo y Frontera: estructuras de datos utilizadas para la exploración, generación del árbol de búsqueda y construcción de soluciones, sea cual sea la estrategia de búsqueda utilizada.
4. Profundidad máxima: nivel de profundidad límite del árbol de búsqueda. Puesto que en las estrategias de Anchura, Costo Uniforme y Profundidad Simple esta profundidad no es finita se simulará esta situación asumiendo una profundidad máxima, por ejemplo, igual a 99.999. Para las estrategias de Profundidad Acotada e Iterativa este valor de profundidad será definido por el usuario previamente a la ejecución del programa.
5. Incremento para la profundidad: En la estrategía de Profundidad Iterativa el usuario deberá definir cuál será el incremento de la Profundidad Máxima una vez que el algoritmo de búsqueda haya explorado todos los nodos del árbol con profundidad igual a la Profundidad Máxima establecida.
6. Salida del programa: si el algoritmo de búsqueda encuentra algún nodo con un estado igual al estado objetivo, la secuencia de acciones que permiten alcanzar la solución se almacenará en un fichero de texto en el que se registrará una acción/movimiento por línea.

2 - Algoritmo básico de búsqueda

Para la implementación del algoritmo básico de búsqueda unificado sea cual sea la estrategia utilizada, se han añadido nuevos métodos y modificados otros ya existentes de la clase

Problema y uno de los constructores de la Clase Nodo. A continuación se detallan los algoritmos implementados en las clases mencionadas.

2.1 - Problema

2.1.1 - busqueda()

Entre todas las estrategias de búsqueda a implementar, la estrategia de búsqueda iterativa supone un caso particular de la estrategía de Profundidad Acotada. En la Iterativa, si el árbol de búsqueda ha analizado todos los nodos con profundidad menor o igual a la profundidad máxima establecida y ningún nodo verifica la función esObjetivo(), entonces relanza de nuevo el algoritmo de *búsqueda_acotada()* con una profundidad límite igual a la profundidad máxima con la que se invocó el algoritmo por última vez más un incremento de cota. El *Listado 24* muestra la implementación de esta estrategia mediante un bucle while que itera mientras no se encuentre solución y la profundidad actual sea menor que un valor límite establecido.

Para el resto de estrategias, es decir Anchura, Coste Uniforme, Profundidad y Profundidad Acotada, el método *búsqueda_acotada()* únicamente deberá ser invocado una única vez. Para ello, el valor de la variable *inc_cota* se igualará al valor de la variable *prof_max* (99.999) asegurando que no se realizará una segunda iteración del bucle while en los casos en los que no se obtenga solución tras la ejecución de *busqueda_acotada(),* es decir, n sea igual a null.

```
public Nodo busqueda(int estrategia, int prof_max, int inc_cota) throws IOException{
        Nodo n = null;
        int cota_act = inc_cota;
        while (n == null && cota_act <= prof_max) {
                n = busqueda_acotada(estrategia, cota_act);
                cota_act += inc_cota;
        }
        tfin=Stat.obtenerTiempo('M');
        ttotal = tfin-tinicio;
        return n;
}
```

Listado 23: *implementación método busqueda()*

2.1.2 - busqueda_acotada()

Como se observa en el *Listado 23* este método es el encargado de lanzar el algoritmo de búsqueda, atendiendo a la estrategia y profundidad máxima. El procedimiento que sigue el algoritmo es el siguiente:

1. En primer lugar, se crea la Frontera y se inserta el nodo con el estado inicial del problema.
2. Si la Frontera no está vacía y no se ha encontrado solución, se extrae el Nodo situado en la cabeza de la Frontera eliminandolo de ésta, garantizado el método compareTo de la clase Nodo que el nodo extraído es el de menor valor. En caso contrario finaliza la ejecución del método busqueda_acotada(), devolviendo el nodo que cumple el test objetivo en caso de existir solución, o un valor *null* en caso de no haberla encontrado.
3. Cada vez que se extrae un nodo de la Frontera, se comprueba si el estado del Nodo seleccionado es un estado objetivo. En caso afirmativo terminaría la ejecución. En caso contrario, se generan los sucesores del Nodo seleccionado mediante la llamada al método *funcionSucesor(Nodo n_actual.getEstado)* del EspacioDeEstados y continúa la ejecución por el punto 4.

4. Si se han generado nuevos sucesores a partir del Nodo seleccionado, a continuación se generan los nuevos Nodos a incorporar en la Frontera mediante la llamada al constructor *Nodo(Nodo parent, Character accion, Puzzle sucesor, String estrategia, int prof_max)* de la *clase Nodo*. El constructor de la clase Nodo tendrá la responsabilidad de controlar si la profundidad de los nuevos nodos no excede al parámetro *prof_max* (en cuyo caso lo indicará poniendo el atributo parent del nodo a *null*) y de asignar los valores de los nodos en función de la estrategia.
5. Los Nodos sucesores cuyo padre no sea *null* (no exceden la profundidad máxima) se insertan en la frontera y se vuelve al punto 2.

En el caso en el que el algoritmo de búsqueda alcance el valor máximo de la memoria Heap de la Máquina Virtual de Java, ésta lanzará una excepción de tipo OutOfMemoryError que provocará la finalización de la ejecución del algoritmo sin haber encontrado solución e informará al usuario de esta situación.

```
public Nodo busqueda_acotada(String estrategia, int prof_max){
boolean solucion = false;
Nodo n_actual = null;
Frontera1 frontera=Frontera1.crearFrontera();
Nodo n_inicial=new Nodo(getEinicial());
if(tinicio == 0) tinicio = Stat.obtenerTiempo('M');
frontera.insertar(n_inicial);
System.out.println("Generando arbol...");
try {
        while(!solucion && !frontera.esVacia()){
                n_actual = frontera.eliminar();
                if(Estado_Meta(n_actual.getEstado())) solucion=true;
                else{
                ArrayList<Sucesor> sucesores=getEde().funcionSucesor(n_actual.getEstado());
                        for(int i=0; i<sucesores.size(); i++){
                                Nodo n = new Nodo(n_actual, sucesores.get(i).getAccion(),
sucesores.get(i).getEstado(), estrategia, prof_max);
                                if(n.getParent()!=null) frontera.insertar(n);
                        }
                }
        }
        System.out.println("Arbol generado");
}catch(OutOfMemoryError e){
        System.out.println("ERROR: Limite de memoria alcanzada. Solución no encontrada");
        return null;
}
if(solucion){
        System.out.println("¡Solución encontrada!");
        return n_actual;
}
else{
        System.out.println("¡No se ha encontrado solución!");
        return null;
}
}
```

Listado 22: *implementación método busqueda()*

2.1.4 - escribirSolucion()

Una vez encontrado un Nodo cuyo estado cumple la condición *esOjetivo()*, deberá escribirse en un fichero la secuencia de acciones que nos permiten alcanzar la solución. Para obtener este fichero de secuencia de acciones nos hemos apoyado en una estructura de datos tipo ArrayList() como puede apreciarse en el *Listado 24*. Esto es necesario debido a que las referencias entre nodos se realiza de hijos a padres por lo que si no se realiza esa inversión el conjunto de acciones nos conduciría desde el estado objetivo al estado inicial.

Además de la solución del problema, en el fichero de texto se muestran también los nombres de los ficheros original y modificado, la estrategia de búsqueda utilizada, el número de filas y columnas, y el tiempo que ha llevado encontrar la solución.

Este método recibe por parámetro el nombre de la estrategia, la cual posteriormente se añadirá al nombre del fichero solución.txt. De esta manera quedará un fichero con el nombre solucion_[estrategia].txt.

```
public void escribirSolucion(Nodo n, String estrategia, String path_original, String
path_modificado, int rows, int cols) throws IOException{
		FileWriter fwi=new FileWriter("solucion_"+estrategia+".txt");
		ArrayList <Nodo> solucion=new ArrayList <Nodo>();
		fwi.write("Ruta del archivo original: "+path_original+"\n");
		fwi.write("Ruta del archivo modificado: "+path_modificado+"\n");
		fwi.write("Número de filas: "+rows+"\n");
		fwi.write("Número de columnas: "+cols+"\n");
		fwi.write("Estrategia: "+estrategia+"\n");
		fwi.write("Tiempo empleado en encontrar la solución: ");
		if(ttotal/Math.pow(10, 9)>1)
			fwi.write("Tiempo empleado en encontrar la solución:
"+ttotal/Math.pow(10, 9)+"s\n");
		else{
			if (ttotal/Math.pow(10, 6)>1)
				fwi.write("Tiempo empleado en encontrar la solución:
"+ttotal/Math.pow(10, 6)+"ms\n");
			else
				fwi.write("Tiempo empleado en encontrar la solución:
"+ttotal+"ns\n");
		}
		fwi.write("Profundidad a la que se alcanza un nodo
objetivo:"+n.getProfundidad()+"\n\n");
		fwi.write("Acciones llevadas a cabo para solucionar el puzzle:\n");

		while(n.getParent()!=null){
			solucion.add(n);
			n=n.getParent();
		}
		for (int i=solucion.size()-1;i>=0;i--){
			fwi.write("Movimiento: "+solucion.get(i).getAccion()+"\n");
		}

		fwi.close();
	}
```

Listado 24: *implementación método busqueda()*

2.2 - Nodo

En el *Listado 25* se muestra la nueva implementación del constructor de la Clase Nodo. Tras la modificación, se puede observar cómo en función de la estrategia utilizada la función de evaluación de los nodos son diferentes según:

1. Anchura: la función de evaluación del Nodo expandido es igual a la profundidad del Nodo padre más una unidad
2. Costo Uniforme: la función de evaluación del Nodo expandido es igual al costo del Nodo padre más una unidad
3. Profundidad Simple, Acotada e Iterativa, la función de evaluación es igual al opuesto de la profundidad del Nodo padre incrementada en una unidad
4. A* : la función de evaluación del Nodo expandido es igual a la profundidad del Nodo padre más una unidad más la heurística.

Además, como se mencionó anteriormente, si la profundidad de un nodo excede el valor de prof_max se le notifica al método búsqueda igualando el valor del atributo parent del nodo a *null*.

```
public Nodo(Nodo parent, Character accion, Estado sucesor, String estrategia, int
prof_max){
		this.parent = parent;
		this.profundidad = this.parent.getProfundidad()+1;
		this.costo = this.parent.getCosto()+1;
		this.accion = accion;
		this.estado = sucesor;
		if(estrategia == "Anchura")
			this.valor = this.parent.getProfundidad()+1;
		else if(estrategia == "Costo Uniforme")
			this.valor =  this.parent.getCosto()+1;
		else if(estrategia == "Profundidad" || estrategia == "Profundidad
Acotada" || estrategia == "Profundidad Iterativa")
			this.valor = (-1)*(this.parent.getProfundidad()+1);
		else if(estrategia == "A estrella")
			this.valor = (this.parent.getProfundidad()+1) +
this.estado.heuristica();
		if(this.profundidad > prof_max)
			this.parent = null;
	}
```

Listado 25: *implementación método busqueda()*

3. Interfaz Gráfica

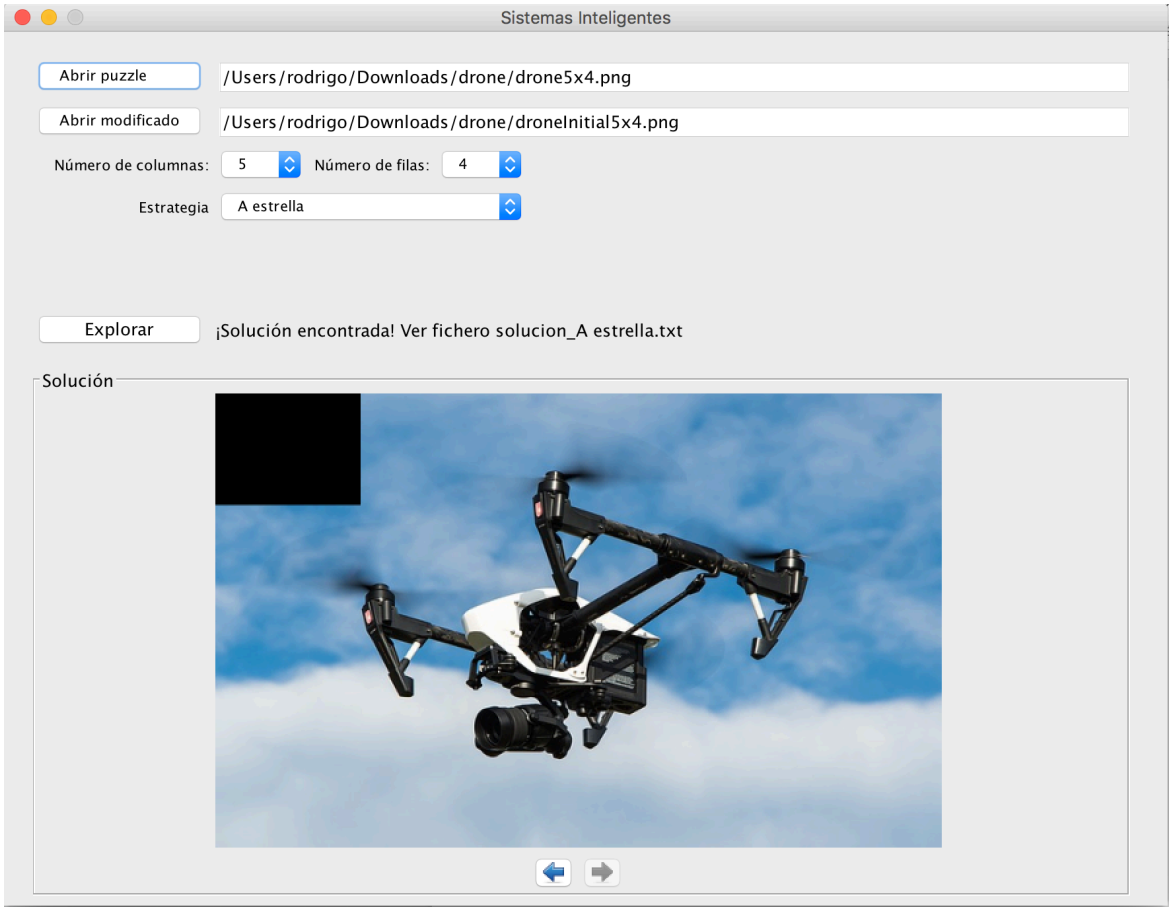

Figura 5*: Interfaz Gráfica de Usuario*

Con el fin de que la introducción de parámetros por parte del usuario sea más atractiva e intuitiva, se ha decidido diseñar una interfaz gráfica como la mostrada en la *Figura 5*. En dicha interfaz el usuario seleccionará el archivo original, el archivo modificado, el número de filas y columnas y la estrategia a usar para la búsqueda de la solución del puzzle. En el caso en el que se haya seleccionado la estrategia de Profundidad Acotada deberá introducirse también un valor de profundidad máxima y, si la estrategia seleccionada es Profundidad Iterativa, el incremento de cota deseado.

Además de esto, el usuario puede visualizar paso a paso y mediante imágenes (pudiendo avanzar y retroceder cuando lo desee) el camino seguido en la construcción de la solución desde el estado inicial hasta el estado objetivo.