

SISTEMAS INTELIGENTES - PRÁCTICA DE LABORATORIO

# DOCUMENTACIÓN

## Generación de un Laberinto y su Árbol de Búsqueda

**AUTORES:**

SERGIO CAÑADILLAS PÉREZ

JAIRO CELADA CEBRIAN

CARLOS MORENO MAROTO

**GRADO DE INGENIERÍA INFORMÁTICA**

**CURSO ACADÉMICO 2020/2021**

**CONVOCATORIA ORDINARIA**

## FICHA del TRABAJO:

**Código:** Práctica de Laboratorio      **Fecha:** 29/11/2020

**Título:** Generación de un Laberinto y su Árbol de Búsqueda

### EQUIPO:

Nº: B1\_10

Apellidos y Nombre	Firma	Puntos
Cañadillas Pérez, Sergio	05717941A	33.33%
Celada Cebrian, Jairo	09065128T	33.33%
Moreno Maroto, Carlos	05969304E	33.33%



# ÍNDICE

1. **Introducción del Problema y Definición como un Problema de Búsqueda en el Espacio de Estados.** – 1
2. **Tareas realizadas con la Justificación de los Criterios y Opciones tomadas. 3 Subsecciones, una por cada Tarea.** – 4
3. **Manual de Usuario.** – 21
4. **Valoración Personal del Modelo de Práctica.**  
Cada integrante del grupo expresará una valoración general sobre la práctica, valorando los objetivos, la metodología y los resultados obtenidos. – 26
5. **Valoración del equipo.**  
Cada integrante del grupo expresará una valoración a su aportación al desarrollo de la práctica indicando una Valoración Propia y Otra de sus compañeros. – 29



## **1. Introducción del Problema y Definición como un Problema de Búsqueda en el Espacio de Estados.**

Analizando la problemática presentada, se haya una posible solución al Laberinto previamente generado, llegar a ello se consigue mediante una búsqueda en el espacio de estados, proceso en el cual, considerando como sucesivos estados de una instancia, la solución o meta será encontrar un "estado final" con unas características deseadas.

El espacio de estados, se define como un conjunto de estados contenidos en el problema. Según la explicación, el conjunto de estados forma un grafo, donde se encuentran conectados 2 estados si en ellos puede llevarse a cabo una operación que transforme el primero en el segundo.

Con frecuencia el grafo del espacio de estados tiene un peso considerable como para ser generado y guardado en memoria. En consecuencia, los nodos se generan en el momento en el que son explorados. Asimismo, la solución consistirá en el camino desde un estado inicial hasta el estado final/objetivo.

Esta práctica consta de un problema dividido en 3 entregas.

En la primera entrega, se genera un laberinto en el que se introducirán las dimensiones (número de filas y columnas) de este y, mediante la implementación del algoritmo de Wilson, se creará y dará una posible solución. Para ello, la solución será exportada como archivo .json junto a su imagen .jpg, en la ruta del proyecto. De igual manera se podrá importar un archivo .json a través de su ruta, y, la imagen que muestra una posible solución se exportará como parte de las “soluciones propuestas”.

En la segunda entrega, con la tarea 1 completada correctamente, se continuó con la resolución del hito implementando el conjunto de estados, la función sucesor, el estado inicial, el objetivo y el costo. En consecuencia, modificaremos las clases subsecuentes que posibiliten la creación del árbol mediante sus nodos y estados, pero como los susodichos nodos carecen de valor en la acción, profundidad, heurística y valor, solo muestra la estructura del árbol ordenado como una posible solución, exportando el archivo .json a través de la ruta del archivo como en las “soluciones propuestas”.

En la tercera entrega, con la tarea 2 completada correctamente, se realizará un planteamiento del problema y la consecuente resolución mediante diferentes algoritmos de búsqueda. Se presentan estrategias no informadas como: búsqueda en anchura (BREADTH), búsqueda en profundidad acotada (DEPTH) y búsqueda de costo uniforme (UNIFORM). Y por el contrario estrategias informadas como: búsqueda voraz (GREEDY) y búsqueda A\* (A). Siguiendo con lo anterior, cada nodo poseerá su propia acción, profundidad, heurística y valor, dependiendo de la estrategia de búsqueda como solución en la ruta del proyecto, exportando el archivo .json mediante la ruta del archivo y se debe exportar la imagen de la solución del .json importado al igual que en las soluciones propuestas.



## 2. Tareas realizadas con la Justificación de los Criterios y Opciones tomadas. 3 Subsecciones, una por cada Tarea.

### TAREA 1: *GENERACIÓN DEL LABERINTO*

#### DECISIONES DE DISEÑO:

Para el desarrollo del proyecto hemos decidido realizarlo en el lenguaje de programación Java, en el entorno de desarrollo Eclipse 2020-09, ya que es un lenguaje que nos estamos formando desde el inicio en el cual tenemos más conocimiento y experiencia. Para cada actualización subiremos el proyecto a GitHub mediante Git.

[https://github.com/Jairoxd98/Jairoxd98-SI-B1\\_10](https://github.com/Jairoxd98/Jairoxd98-SI-B1_10)

Para ello hemos estructurado la entrega en 3 clases: Cell, Grid y Main.

Las clases Cell y Grid tienen atributos privados para que sean accedidos únicamente por la clase y transcient para ignorar ciertos campos durante la serialización de Java en el JSON (para que no siempre tenga acceso al valor de la variable el archivo). La clase Cell tiene los atributos propios de las celdas del laberinto, y los métodos que se utilizan para obtener información y operar con la información de las celdas del laberinto. La clase Grid tiene los atributos propios del laberinto y los métodos que se utilizan para obtener y evaluar la solución comprobando la semántica y generar el laberinto, generar y leer archivo .json y generar archivo .png.

La clase Main es la clase principal que tiene el menú con el cual el usuario podrá interactuar con el sistema para poder realizar los objetivos requeridos.

La clase Main al ejecutar tiene un menú para que el usuario introduzca la opción:

```
Elige una opción:  
1. Generar laberinto  
2. Exportar imagen  
3. Importar laberinto  
4. Salir
```




## Objetivos:

- a. **Crear un laberinto correcto de un tamaño concreto (número de filas x número de columnas) utilizando el algoritmo de Wilson.**

Si el usuario introduce por teclado la opción 1, el programa le pedirá que introduzca las dimensiones de filas y columnas por teclado:


```
Elige una opción:  
1. Generar laberinto  
2. Exportar imagen  
3. Importar laberinto  
4. Salir  
1  
Elige el numero de filas  
10  
Elige el numero de columnas  
10  
Laberinto creado correctamente, JSON creado correctamente
```


 puzzle\_10x10.json

- b. **Exportar ese laberinto como un fichero JSON correcto y una imagen del mismo.**

Si el usuario introduce por teclado la opción 2, el programa exportará la imagen .png del archivo .json de los datos introducidos por el usuario en el directorio del proyecto:

```
Elige una opción:  
1. Generar laberinto  
2. Exportar imagen  
3. Importar laberinto  
4. Salir  
2  
Imagen exportada correctamente
```

 puzzle\_10x10.json

 puzzle\_10x10.png



**a. Importar un fichero JSON, validar su semántica y crear el laberinto correspondiente.**


Si el usuario introduce por teclado la opción 3, el programa le pedirá la ruta del .json para importar, comprobando la semántica del laberinto:


```
Elige una opcion:
1. Generar laberinto
2. Exportar imagen
3. Importar laberinto
4. Salir
3
Indica la ruta del json
C:\Users\Usuario\eclipse-workspace\B1_10-Laberinto\puzzle_70x70.json
JSON importado correctamente
```

**b. Exportar la imagen del laberinto correcto.**

Si el usuario introduce por teclado la opción 2, el programa exportará la imagen .png del archivo .json que el usuario ha importado:

```
Elige una opcion:
1. Generar laberinto
2. Exportar imagen
3. Importar laberinto
4. Salir
2
Imagen exportada correctamente
```

 puzzle\_70x70.json

 puzzle\_70x70.png

## ***Secciones de Código:***

Con este método generamos el laberinto mediante el algoritmo de Wilson. Lo primero que hacemos es coger una celda aleatoria del laberinto y la tomamos como inicio, luego cogemos otra celda aleatoria que no esté visitada y la tomamos como destino, y de manera aleatoria nos movemos en una dirección a una celda adyacente mientras que esta no esté ya visitada o sea la celda destino, y controlando cada vez que nos movemos que no se salga de los bordes del laberinto. En caso de llegar a una celda visitada evita el bucle guardando la celda destino a la visitada y así volviendo al origen. Así vamos recorriendo el laberinto y marcando las celdas visitadas y guardando donde se encuentran los vecinos tanto de la celda en la que estábamos como de la celda adyacente a la que pasamos. Durante el recorrido y su generación se van poniendo los vecinos a true de las celdas por donde se va realizando el camino, y las celdas que acaban en false son las que forman los muros. Cuando se han visitado todas las celdas del laberinto, finaliza la generación de este.



```

public void generateMaze() {

    int cellsVisited = 0;
    while (this.numberCells != cellsVisited) { /* Cuando todas esten visitadas significa que he terminado

        Cell origin = this.getCellEmpty(); /*Asignamos una celda no visitada a la celda inicial*/

        int rowOriginStart = origin.getX();
        int colOriginStart = origin.getY();

        Cell destiny; //Para la primera iteracion establecemos una celda aleatoria
        if (cellsVisited == 0) {
            destiny = this.getCell();
        } else { // Para las siguientes iteraciones establecemos una celda aleatoria que no este visitada
            destiny = this.getCellNoBlank();
        }

        while (!origin.equals(destiny) && origin.isBlank()) { /*Se repite mientras no llegemos al destino

            int index = this.generaNumeroAleatorio(0, this.mov.length - 1);
            int[] movSelected = this.mov[index];
            String direction = this.id_mov[index];

            if ((origin.getX() + movSelected[0]) >= 0
                && (origin.getX() + movSelected[0]) < this.rows
                && (origin.getY() + movSelected[1]) >= 0
                && (origin.getY() + movSelected[1]) < this.cols) { //Comprueba que no se salga de los

                origin.setDirection(direction);

                origin = this.cellsGrid[origin.getX() + movSelected[0]][origin.getY() + movSelected[1]];
            }
        }
    }

    if (!origin.isBlank()) { /*Si llega a una celda visitada
        destiny = origin;
    }

    origin = this.cellsGrid[rowOriginStart][colOriginStart];
    /* Cuando a llegado al destino vamos guardando los vecinos
    * la celda origen y luego en la del vecino visitado resp
    * visitadas durante la generacion del laberinto
    */
    while (!origin.equals(destiny)) {

        cellsVisited++;
        String direction = origin.getDirection();

        boolean neighbors[] = origin.getNeighbors(); // Cogemos
        Cell nextCell = null;
        int[] mov = null;
        switch (direction) { /* Desde la celda actual marcamos
            case "N":
                neighbors[0] = true;
                mov = this.mov[0];
                break;
            case "E":
                neighbors[1] = true;
                mov = this.mov[1];
                break;
            case "S":
                neighbors[2] = true;
                mov = this.mov[2];
                break;
            case "O":
                neighbors[3] = true;
                mov = this.mov[3];
                break;
        }
    }
}

```





```

origin.setNeighbors(neighbors);
origin.setBlank(false);

nextCell = this.cellsGrid[origin.getX() + mov[0]][origin.getY() + mov[1]];
neighbors = nextCell.getNeighbors(); // Cogemos sus vecinos, por si ya tien
switch (direction) { /* Desde la celda adyacente marcamos el vecino de dond
    case "N": //El sur del vecino es el norte de la celda adyacente
        neighbors[2] = true;
        break;
    case "E": //El este del vecino es el oeste de la celda adyacente
        neighbors[3] = true;
        break;
    case "S": //El norte del vecino es el sur de la celda adyacente
        neighbors[0] = true;
        break;
    case "O": //El oeste del vecino es el este de la celda adyacente
        neighbors[1] = true;
        break;
}

nextCell.setNeighbors(neighbors);
origin = nextCell;
}

if (origin.isBlank()) { //Si la celda siguiente no esta ya visitada la marcamos
    cellsVisited++;
}

origin.setBlank(false);
}

```

Este método es para generar (exportar) y escribir la solución del laberinto en un archivo JSON .json

```

public void generateJSON() throws IOException {

    Gson gson = new Gson();

    this.generateCells();

    BufferedWriter bw = new BufferedWriter(new FileWriter("puzzle_" + this.rows + "x" + this.cols + ".json"));

    bw.write(gson.toJson(this));

    bw.close();
}

```



Este método es para generar una imagen de un json. El tamaño sobre lo que se dibuja el laberinto de la imagen varía en función de las dimensiones de las filas y columnas. Comprueba en cada celda si tiene línea en el norte, este, sur y oeste y si no tiene dibuja dos puntos en el eje de coordenadas y dibuja una línea (muro).

```
public void exportToIMG() {  
  
    BufferedImage imagen = new BufferedImage(this.cols * 30, this.rows * 30, BufferedImage.TYPE_INT_RGB);  
  
    Graphics2D g = imagen.createGraphics();  
  
    for (int i = 0; i < this.cellsGrid.length; i++) { /* Recorremos las celads del laberinto */  
        for (int j = 0; j < this.cellsGrid[0].length; j++) {  
            boolean[] n = this.cellsGrid[i][j].getNeighbors(); /*Cogemos los vecinos de la celda*/  
            /*Si tiene un muro en el Norte seleccionamos el punto de la esquina superior izquierda de la celda*/  
            if (!n[0]) {  
                g.drawLine((20 * j) + 10, 20 * (i + 1), (20 * j) + 30, 20 * (i + 1));  
            }  
            /*Si tiene un muro en el Este seleccionamos el punto de la esquina superior derecha de la celda*/  
            if (!n[1]) {  
                g.drawLine((20 * j) + 30, 20 * (i + 1), (20 * j) + 30, 20 * (i + 2));  
            }  
            /*Si tiene un muro en el Sur seleccionamos el punto de la esquina inferior izquierda de la celda*/  
            if (!n[2]) {  
                g.drawLine((20 * j) + 10, 20 * (i + 1) + 20, (20 * j) + 30, 20 * (i + 1) + 20);  
            }  
            /*Si tiene un muro en el Oeste seleccionamos el punto de la esquina superior izquierda de la celda*/  
            if (!n[3]) {  
                g.drawLine((20 * j) + 10, 20 * (i + 1), (20 * j) + 10, 20 * (i + 2));  
            }  
        }  
    }  
  
    try {  
        ImageIO.write(imagen, "png", new File("puzzle_" + this.rows + "x" + this.cols + ".png"));  
    } catch (IOException e) {  
        System.out.println("Error de escritura");  
    }  
}
```



Este método se utiliza para cuando se exporta un JSON, para ello hay que quitar algunos elementos de la llave del TreeMap y cambiarle el formato.

```
public void generateCellsGrids() throws Exception {

    this.numberCells = this.rows * this.cols;
    this.max_n = 4;

    this.cellsGrid = new Cell[this.rows][this.cols];

    for (Map.Entry<Object, Object> entry : this.cells.entrySet()) { /* Recorre
        String key = (String) entry.getKey();
        LinkedHashMap value = (LinkedHashMap) entry.getValue();
        /*Reemplazamos la sintaxis necesaria para extraer los datos de la llave
        key = key.replace("(", " ").trim();
        key = key.replace(")", " ").trim();
        key = key.replace(" ", "").trim();

        String[] parts = key.split(","); //Partimos en "x" e "y"
        /*Tomamos los calores como enteros*/
        int x = Integer.parseInt(parts[0]);
        int y = Integer.parseInt(parts[1]);

        Cell c = new Cell(x, y, false);
        ArrayList<Boolean> n = (ArrayList<Boolean>) value.get("neighbors");

        boolean[] neighbors = new boolean[4];

        for (int i = 0; i < neighbors.length; i++) { //Rellenamos los vecinos
            neighbors[i] = n.get(i);
        }

        c.setNeighbors(neighbors);

        this.cellsGrid[x][y] = c;
    }

    checkCells(); //Llamada al metodo checkCells para comprobar si la semántica
}
```



Este método se utiliza para comprobar la semántica del laberinto de un json importado y comprueba que tiene una estructura correcta. Recorre y controla que el json tenga los muros de las celdas del laberinto (en la parte superior en el norte, en la parte derecha en el este, en la parte de abajo en el sur, en la parte izquierda en el oeste y si tiene en todas las direcciones). También controla que si una celda tiene un vecino en el norte también el vecino adyacente tenga vecino en el sur, si una celda tiene un vecino en el sur también el vecino adyacente tenga vecino en el norte, si una celda tiene un vecino en el este también el vecino adyacente tenga vecino en el oeste y si una celda tiene un vecino en el oeste también el vecino adyacente tenga vecino en el este.

```
private void checkCells() throws Exception {
    for (int i = 0; i < this.cellsGrid.length; i++) { /*Recorremos las celdas del laberinto*/
        for (int j = 0; j < this.cellsGrid[0].length; j++) {
            Cell c = this.cellsGrid[i][j];
            if (i == 0 && c.getNeighbors()[0]) { //Si es de la fila 0 y no tiene un muro al Norte, el JSON es incorrecto
                throw new Exception();
            }
            if (i == (this.cellsGrid.length - 1) && c.getNeighbors()[2]) { //Si es de la ultima fila y no tiene un muro al Sur, el JSON es incorrecto
                throw new Exception();
            }
            if (j == 0 && c.getNeighbors()[3]) { //Si es de la columna 0 y no tiene un muro al oeste, el JSON es incorrecto
                throw new Exception();
            }
            if (j == (this.cellsGrid[0].length - 1) && c.getNeighbors()[1]) { //Si es de la ultima columna y no tiene un muro al este, el JSON es incorrecto
                throw new Exception();
            }
            if (!c.getNeighbors()[0] && !c.getNeighbors()[1] && !c.getNeighbors()[2] && !c.getNeighbors()[3]) {
                throw new Exception();
            }
        }
    }

    for (int k = 0; k < c.getNeighbors().length; k++) {
        if (c.getNeighbors()[k]) {
            switch (k) {
                case 0: /*Comprobamos que si hay un muro al norte, la celda al norte de esta tiene un muro al sur*/
                    if (!this.cellsGrid[i + this.mov[k][0]][j + this.mov[k][1]].getNeighbors()[2]) {
                        throw new Exception();
                    }
                    break;
                case 1: /*Comprobamos que si hay un muro al este, la celda al este de esta tiene un muro al oeste*/
                    if (!this.cellsGrid[i + this.mov[k][0]][j + this.mov[k][1]].getNeighbors()[3]) {
                        throw new Exception();
                    }
                    break;
                case 2: /*Comprobamos que si hay un muro al sur, la celda al sur de esta tiene un muro al norte*/
                    if (!this.cellsGrid[i + this.mov[k][0]][j + this.mov[k][1]].getNeighbors()[0]) {
                        throw new Exception();
                    }
                    break;
                case 3: /*Comprobamos que si hay un muro al oeste, la celda al oeste de esta tiene un muro al este*/
                    if (!this.cellsGrid[i + this.mov[k][0]][j + this.mov[k][1]].getNeighbors()[1]) {
                        throw new Exception();
                    }
                    break;
            }
        }
    }
}
```



## **TAREA 2: DEFINICIÓN DEL PROBLEMA Y LAS ESTRUCTURAS PARA EL ÁRBOL DE BÚSQUEDA**

### **DECISIONES DE DISEÑO:**

Para ello hemos estructurado la entrega en 5 clases: Cell, Grid, Main, Estado y Nodo. La clase Cell no se ha actualizado, tiene la información requerida en la tarea 1.

La clase Grid no se ha actualizado, tiene la información requerida en la tarea 1.

La clase Estado contiene los atributos propios del mismo, y las clases utilizadas para obtener y operar con la información de este.

La clase Nodo contiene los atributos propios del mismo, y las clases utilizadas para obtener y operar con la información de este.

La clase Main es la clase principal que tiene el menú con el cual el usuario podrá interactuar con el sistema para poder realizar los objetivos requeridos de la entrega de la tarea 1 y la tarea 2.

Como solución utilizamos una PriorityQueue de Java. Su finalidad es la ordenación de los objetos en función de su prioridad. En nuestro caso, primero compara y ordena de menor a mayor por el valor del nodo, en caso de este ser igual, la ordenación será por fila y columna.

Al ejecutar, la clase Main muestra un menú en el que el usuario introduce una de las distintas opciones:

```
Elige una opción:  
1. Generar laberinto  
2. Exportar imagen  
3. Importar laberinto  
4. Exportar Hito 2  
5. Salir
```



## Comprobaciones:

1. Construir manualmente los json para generar y leer problemas (JSON) con los laberintos suministrados en la tarea anterior.
2. Generar de forma aleatoria artefactos nodos e Insertar dichos nodos en la frontera para comprobar su inserción correcta.

```
Elige una opcion:
1. Generar laberinto
2. Exportar imagen
3. Importar laberinto
4. Exportar Hito 2
5. Salir
4
Indica la ruta del archivo .json
src//Laberinto//sucesores_10x10.json
JSON importado correctamente

(0, 0) (9, 9) sucesores_10X10_maze.json
Nodo [10][0, [2, 2], null, accion, 0.0, 0.0, 4.0]
Nodo [5][0, [3, 1], null, accion, 0.0, 0.0, 4.0]
Nodo [8][0, [3, 2], null, accion, 0.0, 0.0, 4.0]
Nodo [6][0, [4, 3], null, accion, 0.0, 0.0, 4.0]
Nodo [7][0, [1, 1], null, accion, 0.0, 0.0, 3.0]
Nodo [9][0, [1, 4], null, accion, 0.0, 0.0, 3.0]
Nodo [4][0, [1, 4], null, accion, 0.0, 0.0, 3.0]
Nodo [6][0, [2, 1], null, accion, 0.0, 0.0, 3.0]
Nodo [6][0, [3, 1], null, accion, 0.0, 0.0, 3.0]
```

## Secciones de Código:

El “case 4:” se encuentra en la clase “Main”, esta opción del menú es utilizada para, mediante la lectura de un .json, crear la frontera con el siguiente método y los nodos requeridos.

```
case 4: /*Opcion para crear la frontera y sus nodos mediante un .json*/

String jsonCont = getJSON(askJSON()); //Obtenemos el contenido del JSON

JsonParser parser = new JsonParser();
JsonObject gsonObj = parser.parse(jsonCont).getAsJsonObject();

String initial = gsonObj.get("INITIAL").getString(); //Nodo Inicio
String objective = gsonObj.get("OBJECTIVE").getString(); //Nodo Objetivo
String maze = gsonObj.get("MAZE").getString(); //Nombre del .json a utilizar

try {
    String jsonContent = getJSON("src//Laberinto//"+maze);/* PONER DONDE SE ENCUENTRA */
    Gson gson = new Gson();

    grid = gson.fromJson(jsonContent, Grid.class); //Extrae el contenido del JSON que pedimos por teclado
    grid.generateCellsGrids(); //Genera el laberinto mediante los datos del JSON importado
    System.out.println("JSON importado correctamente\n");
} catch (Exception ex) {
    grid=null;
    System.out.println("JSON no compatible\n");
}
System.out.println(initial+" "+objective+" "+maze);
generarFrontera(grid);
break;
```



El método “generarFrontera” se encuentra en la clase “Main”, y es llamado a través de la interfaz de menú eligiendo la opción 4, en él, instanciamos una PriorityQueue<Nodo> llamada “frontera” en la cual introducimos de manera ordenada por prioridad de los objetos y mediante un “for”, 20 nodos que a continuación imprimimos por pantalla.

```
public static void generarFrontera (Grid g) {
    PriorityQueue<Nodo> frontera = new PriorityQueue<Nodo>();

    for (int i = 0; i<20; i++) {
        frontera.add(new Nodo(null, new Estado(((int) (Math.random()*4+1)), ((int) (Math.random()*4+1)), "e" ,
            ((int) (Math.random()*10+1)), 0, "accion", 0, 0/*heurística por definir*/, ((int) (Math.random()*4+1))));
    }

    for (int i = 0; i<20; i++) {
        System.out.println(frontera.poll().toString());
    }
}
```

El método “funcionSucesores” se encuentra en la clase “Main”, y será instanciado desde métodos futuros con intención de comprobar los estados de Norte, Este, Sur y Oeste al intentar dirigirse en esa dirección:

```
private static ArrayList<Estado> funcionSucesores (Estado e, Grid g){
    ArrayList<Estado> list = new ArrayList<Estado>();

    if (e.getId()[0] != 0 && g.getCellsGrid()[e.getId()[0]][e.getId()[1]].getNeighbors()[0]) { //N (comprobar el estado de ir hacia el Norte)
        list.add(new Estado(g.getId_mov()[0], e.getId()[0], e.getId()[1]+1, 1));
    }
    if (e.getId()[1] != g.getCols()-1 && g.getCellsGrid()[e.getId()[0]][e.getId()[1]].getNeighbors()[1]) { //E (comprobar el estado de ir hacia el Este)
        list.add(new Estado(g.getId_mov()[1], e.getId()[0]+1, e.getId()[1], 1));
    }
    if (e.getId()[0] != g.getRows()-1 && g.getCellsGrid()[e.getId()[0]][e.getId()[1]].getNeighbors()[2]) { //S (comprobar el estado de ir hacia el Sur)
        list.add(new Estado(g.getId_mov()[2], e.getId()[0], e.getId()[1]-1, 1));
    }
    if (e.getId()[0] != 0 && g.getCellsGrid()[e.getId()[0]][e.getId()[1]].getNeighbors()[3]) { //O (comprobar el estado de ir hacia el Oeste)
        list.add(new Estado(g.getId_mov()[3], e.getId()[0]-1, e.getId()[1], 1));
    }
    return list;
}
```

El método “esObjetivo” se encuentra en la clase “Main”, su función es comprobar que hemos llegado al nodo objetivo/correcto.

```
public static boolean esObjetivo (Nodo n, Grid g) {
    if (n.getEstado().getId()[0] == g.getRows()-1 && n.getEstado().getId()[1] == g.getCols()-1) {
        return true;
    } else return false;
}
```

El siguiente método “compareTo” se encuentra en la clase “Nodo”, este es utilizado para comparar el valor del Nodo (f), y por consiguiente mantener un orden de mayor a menor, ya que, en un árbol de búsqueda, accederemos siempre al nodo con menor valor de su función.

```
public int compareTo (Nodo n) {
    if (this.getF() > n.getF()) {
        return -1;
    } else if (this.getF() < n.getF()) {
        return 1;
    } else {
        return this.getEstado().compareTo(n.getEstado());
    }
}
```



El método “toString()” de la clase “Nodo”, mantiene el formato requerido en la especificación.

```
public String toString() {  
    return "Nodo [" + id + "]" + costo + ", [" + estado.getId()[0] + ", " + estado.getId()[1] + "], "  
    + ((padre != null)?padre.getId():"null") + ", " + accion + ", " + d + ", " + h + ", " + f + "];"  
}
```

A continuación, encontramos el método “compareTo” en la clase “Estado”, este es utilizado para comparar por filas y columnas, en un orden de menor a mayor, ya que, en un laberinto, tomamos la celda de arriba a la izquierda (0, 0) como inicial, y la de abajo a la derecha (n-1, n-1) como celda final.

```
public int compareTo (Estado e) {  
    if (this.getId()[0] < e.getId()[0]) {  
        return -1;  
    } else if (this.getId()[0] > e.getId()[0]) {  
        return 1;  
    } else  
        return (this.getId()[1] > e.getId()[1])?1:-1;  
}
```

En la en la clase “Estado” hallaremos el método “generateMD5”, con intención de referenciar un nodo de forma única, aunque varios tengan un mismo estado.

```
public static String generateMD5(String id) throws NoSuchAlgorithmException {  
  
    String IdMD5;  
  
    MessageDigest md = MessageDigest.getInstance("MD5");  
    md.update(id.getBytes());  
    byte[] digest = md.digest();  
    StringBuffer sb = new StringBuffer();  
    for (byte b : digest) {  
        sb.append(String.format("%02x", b & 0xff));  
    }  
    IdMD5 = sb.toString();  
  
    return IdMD5;  
}
```





## TAREA 3: LABERINTOS INFORMADOS Y ALGORITMO DE BÚSQUEDA

### DECISIONES DE DISEÑO:

Para ello hemos estructurado la entrega en 5 clases: Cell, Grid, Main, Estado y Nodo.

La clase Cell contiene los atributos propios del mismo, y las clases utilizadas para obtener y operar con la información de este.

La clase Grid contiene los atributos propios del mismo, y las clases utilizadas para obtener y operar con la información de este.

La clase Estado contiene los atributos propios del mismo, y las clases utilizadas para obtener y operar con la información de este.

La clase Nodo contiene los atributos propios del mismo, y las clases utilizadas para obtener y operar con la información de este.

La clase Main es la clase principal que tiene el menú con el cual el usuario podrá interactuar con el sistema para poder realizar los objetivos requeridos de la entrega de las tareas 1, 2 y 3.

Al ejecutar, la clase Main muestra un menú en el que el usuario introduce una de las distintas opciones:

Elige una opcion:

1. Generar laberinto
2. Exportar imagen
3. Importar laberinto
4. Importar JSON
5. Encontrar una solución mediante Árbol de Búsqueda
6. Salir

### Comprobaciones:

Elige una opcion:

1. Generar laberinto
2. Exportar imagen
3. Importar laberinto
4. Importar JSON
5. Encontrar una Solucion mediante arbol de Busqueda
6. Salir

4

Indica la ruta del archivo .json

src/Laberinto/problema\_5x5.json

JSON importado correctamente



Elija la estrategia para implementar el problema

1. Anchura
2. Profundidad
3. Costo Uniforme
4. Búsqueda Voraz
5. Búsqueda A\*
6. Salir

2

```
[id][cost, state, father_id, action, depth, h, value]
Nodo [0][0, (0, 0), None, None, 0, 8, 1.0]
Nodo [1][3, (1, 0), 0, S, 1, 7, 0.5]
Nodo [3][5, (1, 1), 1, E, 2, 6, 0.3333333333333333]
Nodo [5][9, (0, 1), 3, N, 3, 7, 0.25]
Nodo [9][10, (0, 2), 5, E, 4, 6, 0.2]
Nodo [11][11, (1, 2), 9, S, 5, 5, 0.16666666666666666]
Nodo [14][13, (1, 3), 11, E, 6, 4, 0.14285714285714285]
Nodo [18][15, (1, 4), 14, E, 7, 3, 0.125]
Nodo [23][18, (2, 4), 18, S, 8, 2, 0.11111111111111111]
Nodo [27][19, (2, 3), 23, O, 9, 3, 0.1]
Nodo [31][20, (2, 2), 27, O, 10, 4, 0.09090909090909091]
Nodo [35][21, (2, 1), 31, O, 11, 5, 0.08333333333333333]
Nodo [38][22, (2, 0), 35, O, 12, 6, 0.07692307692307693]
Nodo [41][25, (3, 0), 38, S, 13, 5, 0.07142857142857142]
Nodo [43][26, (3, 1), 41, E, 14, 4, 0.06666666666666667]
Nodo [45][28, (3, 2), 43, E, 15, 3, 0.0625]
Nodo [49][30, (3, 3), 45, E, 16, 2, 0.058823529411764705]
Nodo [53][32, (3, 4), 49, E, 17, 1, 0.05555555555555555]
Nodo [56][36, (4, 4), 53, S, 18, 0, 0.05263157894736842]
```

## Secciones de Código:

Este método se utiliza en la clase Estado para ordenar por filas y columnas. Es utilizado para comparar por filas y columnas, en un orden de menor a mayor, ya que, en un laberinto, tomamos la celda de arriba a la izquierda (0, 0) como inicial, y la de abajo a la derecha (n-1, n-1) como celda final.

```
public int compareTo (Estado e) { //Ordenamos por (filas, columnas)
    if (this.getId()[0] < e.getId()[0]) {
        return -1;
    } else if (this.getId()[0] > e.getId()[0]) {
        return 1;
    } else
        return (this.getId()[1] > e.getId()[1])?1:-1;
}
```



Este método se utiliza en la clase Nodo para ordenar por id y función del nodo. Este es utilizado para comparar el valor del Nodo (f) y el identificador (id), y por consiguiente mantener un orden de mayor a menor, ya que, en un árbol de búsqueda, accederemos siempre al nodo con menor valor de su función.

```
public int compareTo (Nodo n) { //ordenamos de menor a mayor valor (f) y la id del nodo
    if (this.f < n.getF()) {
        return -1;
    }
    else if (this.f > n.getF()) {
        return 1;
    }
    else if (this.estado.getId()[0] < n.estado.getId()[0]) {
        return -1;
    }
    else if (this.estado.getId()[0] > n.estado.getId()[0]) {
        return 1;
    }
    else if (this.estado.getId()[1] < n.estado.getId()[1]) {
        return -1;
    }
    else if (this.estado.getId()[1] > n.estado.getId()[1]) {
        return 1;
    }
    else if (this.id < n.id) {
        return -1;
    }
    else if (this.id > n.id) {
        return 1;
    }
    else {
        return this.getEstado().compareTo(n.getEstado());
    }
    //return 0;
}
```

Este método se utiliza en la clase Main para dar la solución a cada estrategia.

```
private static ArrayList<Nodo> nodoSucesores (Nodo n, Grid g, String estrategia, int id){
    ArrayList<Nodo> list = new ArrayList<Nodo>();
    for (Estado a: funcionSucesores(n.getEstado(), g)) {
        switch (estrategia) {
            case "BREADTH":
                list.add(new Nodo(n, a, id, n.getCosto()+a.getValor(), a.getAccion(), n.getD()+1, calcularHeuristica(a, g), n.getD()));
                break;
            case "DEPTH":
                list.add(new Nodo(n, a, id, n.getCosto()+a.getValor(), a.getAccion(), n.getD()+1, calcularHeuristica(a, g), 1/(n.getD()+1)));
                break;
            case "UNIFORM":
                list.add(new Nodo(n, a, id, n.getCosto()+a.getValor(), a.getAccion(), n.getD()+1, calcularHeuristica(a, g), n.getCosto())); //quito+1
                break;
            case "GREEDY":
                list.add(new Nodo(n, a, id, n.getCosto()+a.getValor(), a.getAccion(), n.getD()+1, calcularHeuristica(a, g), n.getH()));
                break;
            case "A*":
                list.add(new Nodo(n, a, id, n.getCosto()+a.getValor(), a.getAccion(), n.getD()+1, calcularHeuristica(a, g), n.getH()+n.getCosto()));
                break;
        }
        id++;
    }
    return list;
}
```



Este método se utiliza en la clase Main para calcular la heurística mediante la distancia Manhattan.

```
public static int calcularHeuristica (Estado e, Grid g) {  
    return (Math.abs(g.getRows()-e.getId()[0]))+(Math.abs(g.getCols()-e.getId()[1]));  
}
```

Este método se utiliza en la clase Main como auxiliar que se encarga de copiar la información de los nodos.

```
public static Nodo copiarNodo (Nodo n) {  
    Nodo nCopia = new Nodo(n.getPadre(), n.getEstado(), n.getId(), n.getCosto(), n.getAccion(), n.getD(), n.getH(), n.getF());  
    return nCopia;  
}
```

Esta opción del switch de la clase Main es para la tercera entrega en la que se elige por pantalla la estrategia de búsqueda para mostrar la solución.

```
case 5:/* Opcion para encontrar una solucion con las diferente heurísticas*/  
    boolean seguir = true;  
    do {  
        System.out.println("\nElija la estrategia para implementar el problema\n1. Anchura"  
            + "\n2. Profundidad\n3. Costo Uniforme\n4. Busqueda Voraz\n5. Busqueda A*\n6. Salir");  
        option = sc.nextInt();  
        switch (option) {  
            case 1:  
                System.out.println("\t[id][cost,state,father_id,action,depth,h,value]");  
                busqueda(new Estado(inicial[0], inicial[1], null, grid.getCellsGrid()[0][0].getValue(), grid, "BREADTH", 1000000,objetivo);  
                break;  
            case 2:  
                System.out.println("\t[id][cost,state,father_id,action,depth,h,value]");  
                busqueda(new Estado(inicial[0], inicial[1], null, grid.getCellsGrid()[0][0].getValue(), grid, "DEPTH", 1000000,objetivo);  
                break;  
            case 3:  
                System.out.println("\t[id][cost,state,father_id,action,depth,h,value]");  
                busqueda(new Estado(inicial[0], inicial[1], null, grid.getCellsGrid()[0][0].getValue(), grid, "UNIFORM", 1000000,objetivo);  
                break;  
            case 4:  
                System.out.println("\t[id][cost,state,father_id,action,depth,h,value]");  
                busqueda(new Estado(inicial[0], inicial[1], null, grid.getCellsGrid()[0][0].getValue(), grid, "GREEDY", 1000000,objetivo);  
                break;  
            case 5:  
                System.out.println("\t[id][cost,state,father_id,action,depth,h,value]");  
                busqueda(new Estado(inicial[0], inicial[1], null, grid.getCellsGrid()[0][0].getValue(), grid, "A", 1000000,objetivo);  
                break;  
            case 6:  
                seguir = false;  
                break;  
            default:  
                System.out.println("Opcion erronea ");  
                break;  
        }  
    } while (seguir);  
    break;
```



Este método se utiliza en la clase Main para realizar el árbol de búsqueda con las diferentes estrategias.

```
public static ArrayList<Nodo> busqueda (Estado inicial, Grid g, String estrategia, int cota, int []objetivo){
    PriorityQueue<Nodo> frontera = new PriorityQueue<Nodo>();
    ArrayList<Nodo> visitados = new ArrayList<Nodo>();
    Nodo nodo;
    ArrayList<Nodo> sucesores;
    int id = 0; //identificadores de los nodos

    if (estrategia == "profundidad") { //porque en caso de la profundidad, la formula es n/n, y como haga 0/0 peta
        nodo = new Nodo(null, inicial, id, 0, null, 0, 0, 1);
    } else nodo = new Nodo(null, inicial, id, 0, null, 0, 0, 0); //resto de casos

    id++; //al ya hacer el id 0, el siguiente será el id 1
    frontera.add(nodo); //añadimos el nodo inicial

    while (!frontera.isEmpty() && !esObjetivo(frontera.peek(), objetivo)) { //no voy a tratar de buscar si ya he mirado en todo, o si ya he encontrado la solución

        nodo = frontera.poll(); //porque el peek, no lo sacamos, y con poll, lo sacamos y eliminamos de la frontera para no tener que volverlo a mirar
        if (nodo.getD() < cota) { //comprobamos de que no nos hemos pasado del limite de nodo
            sucesores = nodoSucesores(nodo, g, estrategia, id); //miramos los caminos adyacentes
            for (Nodo n: sucesores) { //para cada nodo adyacentes, miramos si lo hemos visitado o no, si no lo he visitado lo meto a la frontera
                if (!visitados.contains(n)) {
                    frontera.add(n);
                    //System.out.println("\t" + n.toString());
                }
                id++; //aumentamos el identificador por cada nodo identificado
                //System.out.println("\t" + n.toString());
            }
        }
        visitados.add(copiarNodo(nodo)); //metemos el nodo actual
    }

    if (!frontera.isEmpty()) { //comprobamos si hay solución o no
        if (esObjetivo(frontera.peek(), objetivo)) { //si es solución entonces procedemos a coger la información y mostrarla
            nodo = frontera.peek();
            ArrayList<Nodo> solucion = new ArrayList<Nodo>();
            while (nodo.getPadre() != null) { //vas cogiendo el nodo actual y lo metes como parte de la solución
                System.out.println("\t" + nodo.toString());
                solucion.add(copiarNodo(nodo));
                nodo = nodo.getPadre();
            }
            System.out.println("\t" + nodo.toString());
            solucion.add(copiarNodo(nodo)); //añadimos el nodo inicial
            return solucion; //agrupación de todos los nodos que hemos ido haciendo
        }
    }
    return null;
}
```



### 3. Manual de Usuario.

Al iniciar el programa a través del botón de Eclipse “Run”, de color verde y situado en la parte de arriba a la izquierda, en la “Console” nos aparecerá el siguiente Menú de opciones, en el que, según la selección de una de ellas (debemos escribir por teclado “1”, “2”, “3”, “4”, “5” o “6”) realizaremos las diferentes acciones del programa.

Elige una opcion:

1. Generar laberinto
2. Exportar imagen
3. Importar laberinto
4. Importar JSON
5. Encontrar una Solucion mediante arbol de Busqueda
6. Salir

En primer lugar, si seleccionamos la primera opción, “1. Generar laberinto”, nos pedirá el número de filas y columnas que querríamos que el Laberinto contuviese, y de forma automática se realizará la acción que describe la opción. Este será almacenado en su ruta por defecto, en este caso en la carpeta del mismo proyecto.

```
1
Elige el numero de filas
5
Elige el numero de columnas
5
Laberinto creado correctamente, JSON creado correctamente
```

A continuación, y ya generado un laberinto con el que poder trabajar, si seleccionamos “2. Exportar imagen”, tendremos la posibilidad de exportar una imagen de este, en la ruta por defecto anteriormente mencionada.

Elige una opcion:

1. Generar laberinto
2. Exportar imagen
3. Importar laberinto
4. Importar JSON
5. Encontrar una Solucion mediante arbol de Busqueda
6. Salir

```
2
Imagen exportada correctamente
```



En el caso de seleccionar la tercera opción, “3. Importar laberinto”, aparecerá el siguiente mensaje en el que se pide la ruta completa de un archivo con la especificación “\_maze”, el cual contiene el .json que define el laberinto, y al pulsar la tecla “Enter”, si la ruta ha sido bien introducida aparecerá el siguiente mensaje:

```
Elige una opcion:
  1. Generar laberinto
  2. Exportar imagen
  3. Importar laberinto
  4. Importar JSON
  5. Encontrar una Solucion mediante arbol de Busqueda
  6. Salir

3
Indica la ruta del archivo .json
B1_10-Laberinto_SI/src/Laberinto/problema_10x10_maze.json
JSON importado correctamente
```

Si la ruta, por algún motivo sintáctico, de “archivo no encontrado”, o una ruta, que aunque lógica, no está bien definida, aparecerá el siguiente mensaje:

```
3
Indica la ruta del archivo .json
Ruta NO Indicada
JSON no compatible
```

En el caso de seleccionar la cuarta opción (“4. Importar JSON”), posibilita que, mediante una ruta definida teclado en la consola, importemos un archivo sin la especificación “\_maze” (esta se añade de manera automática), y de tipo .json. Con él, podremos elegir “2. Exportar imagen”, anteriormente explicada, o “5. Encontrar una Solución mediante árbol de Búsqueda”.

```
Elige una opcion:
  1. Generar laberinto
  2. Exportar imagen
  3. Importar laberinto
  4. Importar JSON
  5. Encontrar una Solucion mediante arbol de Busqueda
  6. Salir

4
Indica la ruta del archivo .json
B1_10-Laberinto_SI/src/Laberinto/problema_5x5.json
JSON importado correctamente
```



Si la ruta fuese incorrecta por algún motivo sintáctico o archivo no encontrado, saltará este error por pantalla:

```
4
Indica la ruta del archivo .json
Ruta NO Indicada
Ruta (El sistema no puede encontrar el archivo especificado)
```

Si la ruta especificada fuese correcta, o lógica, pero no se pudiese encontrar por algún motivo, saltaría este mensaje:

```
4
Indica la ruta del archivo .json
Laberinto/problema_10x10.json
Laberinto\problema_10x10.json (El sistema no puede encontrar la ruta especificada)
```

Después de importar un laberinto mediante las opciones “3. Importar laberinto” o “4. Importar JSON”, tendremos la libertad de seleccionar, exportar una imagen (“2. Exportar imagen”), anteriormente explicada, o Encontrar una Solución mediante árbol de Búsqueda (“5. Encontrar una Solución mediante árbol de Búsqueda”), a través de la quita opción, si seleccionamos esta última, nos imprimirá la matriz que compone el laberinto, y el valor de cada celda. En el ejemplo siguiente imprimimos el “problema\_5x5.json”:

```
Elige una opcion:
    1. Generar laberinto
    2. Exportar imagen
    3. Importar laberinto
    4. Importar JSON
    5. Encontrar una Solucion mediante arbol de Busqueda
    6. Salir

5
0 3 0 2 0
2 1 0 1 1
0 0 0 0 2
2 0 1 1 1
2 0 2 0 3
```

A continuación, el programa nos mostrará un menú con las diferentes estrategias que podemos seguir. Estas definirán el árbol de búsqueda mediante un algoritmo predefinido en el código.





Elija la estrategia para implementar el problema

1. Anchura
2. Profundidad
3. Costo Uniforme
4. Búsqueda Voraz
5. Búsqueda A\*
6. Salir

2

```
[id][cost, state, father_id, action, depth, h, value]
Nodo [0][0, [0, 0], None, None, 0, 8, 1.0]
Nodo [1][3, [1, 0], 0, S, 1, 7, 0.5]
Nodo [3][5, [1, 1], 1, E, 2, 6, 0.3333333333333333]
Nodo [5][9, [0, 1], 3, N, 3, 7, 0.25]
Nodo [9][10, [0, 2], 5, E, 4, 6, 0.2]
Nodo [11][11, [1, 2], 9, S, 5, 5, 0.16666666666666666]
Nodo [14][13, [1, 3], 11, E, 6, 4, 0.14285714285714285]
Nodo [18][15, [1, 4], 14, E, 7, 3, 0.125]
Nodo [23][18, [2, 4], 18, S, 8, 2, 0.11111111111111111]
Nodo [27][19, [2, 3], 23, 0, 9, 3, 0.1]
Nodo [31][20, [2, 2], 27, 0, 10, 4, 0.09090909090909091]
Nodo [35][21, [2, 1], 31, 0, 11, 5, 0.08333333333333333]
Nodo [38][22, [2, 0], 35, 0, 12, 6, 0.07692307692307693]
Nodo [41][25, [3, 0], 38, S, 13, 5, 0.07142857142857142]
Nodo [43][26, [3, 1], 41, E, 14, 4, 0.06666666666666667]
Nodo [45][28, [3, 2], 43, E, 15, 3, 0.0625]
Nodo [49][30, [3, 3], 45, E, 16, 2, 0.058823529411764705]
Nodo [53][32, [3, 4], 49, E, 17, 1, 0.05555555555555555]
Nodo [56][36, [4, 4], 53, S, 18, 0, 0.05263157894736842]
```

En cualquier momento podremos seleccionar la sexta opción (“6. Salir”), por la que, si nos encontramos en el menú de la quita opción (“5. Encontrar una Solución mediante árbol de Búsqueda”), volveremos al anterior menú de Inicio:

Elija la estrategia para implementar el problema

1. Anchura
2. Profundidad
3. Costo Uniforme
4. Búsqueda Voraz
5. Búsqueda A\*
6. Salir

6

Elige una opcion:

1. Generar laberinto
2. Exportar imagen
3. Importar laberinto
4. Importar JSON
5. Encontrar una Solucion mediante arbol de Búsqueda
6. Salir



Y si en el menú de Inicio, seleccionamos la sexta opción (“6. Salir”), saldremos finalmente del programa:

```
6
Has salido del programa
```

Si por un casual, cometemos errores en las interfaces de Menú, nos devolverá a este para facilitar la nueva introducción por teclado:

```
Elige una opcion:
    1. Generar laberinto
    2. Exportar imagen
    3. Importar laberinto
    4. Importar JSON
    5. Encontrar una Solucion mediante arbol de Busqueda
    6. Salir

Opción no válida
Introduce una opcion valida

Elige una opcion:
    1. Generar laberinto
    2. Exportar imagen
    3. Importar laberinto
    4. Importar JSON
    5. Encontrar una Solucion mediante arbol de Busqueda
    6. Salir
```



#### **4. Valoración Personal del Modelo de Práctica.**

**Cada integrante del grupo expresará una valoración general sobre la práctica, valorando los objetivos, la metodología y los resultados obtenidos.**

**SERGIO CAÑADILLAS PÉREZ:**

##### **Práctica**

La práctica que se nos presenta en Sistemas Inteligentes es la aplicación teórica a la práctica, es decir un problema teórico en un problema práctico. Reúne los contenidos teóricos que hemos ido adquiriendo y que son aplicables para el desarrollo e implementación de la práctica. He comprobado se ha ido correspondiendo con la teoría, que durante este tiempo ha habido entregas que nos ha supuesto más trabajo que otras.

##### **Objetivos**

Los objetivos de la práctica es la implementación y resolución de un problema desde la primera entrega con la creación del laberinto hasta la tercera entrega, incrementando la práctica en cada entrega para la resolución de la solución final.

##### **Metodología**

La metodología llevaba en la práctica ha sido buena, debido a que era una práctica interesante que no suponía una carga de trabajo excesiva y tenías que estar activo con los conocimientos adquiridos previamente en la asignatura de estructura de datos y con el uso del lenguaje de programación Java.

##### **Resultados Obtenidos**

Los resultados desde el inicio hasta el final de la práctica han sido los esperados por los miembros del grupo con el aprendizaje individual y mutuo para afrontar este problema y alcanzar los conocimientos requeridos por la asignatura.



## **JAIRO CELADA CEBRIAN:**

### **Práctica**

La práctica está compuesta de varios hitos y un examen final en la que es necesario seguir un seguimiento de la practica constante y aprobar un examen final para poder aprobar la pasa de prácticas.

### **Objetivos**

Los objetivos de la práctica es representar lo aprendido en la teoría en una práctica, en la que además aprendemos por nuestra cuenta a base de buscar información y a prueba y error, al no tener ningún material guiarnos en el aprendizaje más que el enunciado.

### **Metodología**

La metodología me ha parecido un poco estresante ya que teníamos que aprender y estar constantemente con la práctica de inteligentes debido a las entregas constantes a lo largo del curso. Y la realización de un examen final obligatorio en el que si no sacas un 10 no apruebas las practicas no me parece nada correcto ya que, si lo suspendes todo lo esfuerzo a sido para nada, y no entiendo el no poder sacar un 4 ni un 5, tener que sacar directamente un 6 si quieres aprobar la asignatura.

### **Resultados Obtenidos**

Los resultados obtenidos son haber tenido que dedicar mucho tiempo a la realización de la practica en el que sí que he aprendido mucho pero no creo que se vea muy bien reflejado y en una práctica en la que además si tus compañeros no contribuyen a la par que tú te lleva muchísimo tiempo.



## **CARLOS MORENO MAROTO:**

### **Práctica**

La práctica que se nos presenta en Sistemas Inteligentes, es, a mi humilde parecer, un resumen de las distintas disciplinas que a lo largo de los cursos anteriores se ha ido perfeccionando y, que en la problemática presentada podemos pulir, esto ha sido posible a la libertad y ambigüedad de los diferentes hitos, que, al no dar detalles muy específicos de la forma de implementación, posibilitan distintas soluciones ante un mismo problema.

### **Objetivos**

Los objetivos son claros, creación y resolución de problemas, en este caso capacitación de la creación de laberintos que, por medio de un algoritmo, sean resueltos de su estado inicial al final/objetivo.

### **Metodología**

La metodología llevada a cabo para la realización de la práctica me parece que va por buen camino, el fomento de aprendizaje individual, ofrece un rango amplio de presentación de las cualidades y competencias de cada uno.

### **Resultados Obtenidos**

Los resultados obtenidos son concisos. Aprendizaje en organización, desarrollo e implementación de un problema creado y su resolución.



## 5. Valoración del equipo.

**Cada integrante del grupo expresará una valoración a su aportación al desarrollo de la práctica indicando una Valoración Propia y Otra de sus compañeros.**

APELLIDOS Y NOMBRE	VALORACIÓN PROPIA	VALORACIÓN DEL EQUIPO
Cañadillas Pérez, Sergio	Como opinión crítica, he estado trabajando constantemente durante toda la práctica, implementando y ayudando a mis dos compañeros en lo que han necesitado en caso de dudas o problemas. He intentado dedicarle todo el tiempo posible a cada entrega.	<b>Celada Cebrian, Jairo:</b> Ha sido buen compañero compartiendo sus conocimientos, ayudando y trabajando constantemente en las tres entregas.
		<b>Moreno Maroto, Carlos:</b> Ha sido buen compañero compartiendo sus conocimientos, ayudando y trabajando constantemente en las dos últimas entregas.

APELLIDOS Y NOMBRE	VALORACIÓN PROPIA	VALORACIÓN DEL EQUIPO
Celada Cebrián, Jairo:	En esta práctica he tenido que hacer un sobreesfuerzo para poder llevar a cabo los diferentes hitos de la práctica. En la codificación soy el que más ha desarrollado y en las memorias el que menos. Creo que he contestado continuamente dudas en el grupo y que he resuelto muchos problemas y he tenido a mi equipo constantemente al tanto de todo y con todas las explicaciones posibles.	<b>Cañadillas Pérez, Sergio:</b> Al principio tuvimos que hacer el primer hito juntos, posteriormente siguió siendo participativo y se encargó sobre todo de las memorias. Fue constante y trabajador.
		<b>Moreno Maroto, Carlos:</b> En el primer hito no participo, tuvimos que hacerlo Sergio y yo, aunque posteriormente si comenzó a participar. A pesar de la ausencia del principio finalmente si desarrollo más parte de los últimos hitos.



APELLIDOS Y NOMBRE	VALORACIÓN PROPIA	VALORACIÓN DEL EQUIPO
<b>Moreno Maroto, Carlos</b>	<p>Como opinión crítica, tuve mi mayor desempeño en las tareas, 2 y 3, donde dediqué el mayor tiempo a resolver las distintas problemáticas.</p> <p>He trabajado muy duro para sacar adelante el proyecto, a mi humilde parecer.</p>	<p><b>Cañadillas Pérez, Sergio:</b></p> <p>En mi opinión ha trabajado duro desde el primer momento, ha sacado la práctica junto a sus dos compañeros con trabajo y empeño. Volvería a trabajar con él.</p>
		<p><b>Celada Cebrian, Jairo:</b></p> <p>Ha estado en todo momento a la altura, su espíritu crítico y constante ayudó mucho en la realización del proyecto. Volvería a trabajar con él</p>

En Ciudad Real, a Domingo 29 de Noviembre de 2020.

Grado en Ingeniería Informática

