

SISTEMAS INTELIGENTES

SESIÓN 1: *GENERACIÓN DEL LABERINTO*

LABORATORIO: *Grupo B1_10*

COMPONENTES:

Jairo Celada Cebrián

Sergio Cañadillas Pérez

Carlos Moreno Maroto

Índice

Decisiones de diseño	3
Objetivos.....	4
Secciones de código	6

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

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

 puzzle_10x10.json

 puzzle_10x10.png

- c. 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

- d. 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 igualando 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 iteracioens 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 vecino
 * 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();// Cogemo
    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 ce  
            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 celd  
            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 Norte seleccionamos el punto de la esquina superior izquierda de la ce  
            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 valores 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 ir
                throw new Exception();
            }
            if (i == (this.cellsGrid.length - 1) && c.getNeighbors()[2]) { //Si es de la ultima fila y no tiene
                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
                throw new Exception();
            }
            if (j == (this.cellsGrid[0].length - 1) && c.getNeighbors()[1]) { //Si es de la ultima columna y no
                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
                    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 mu
                    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
                    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 m
                    if (!this.cellsGrid[i + this.mov[k][0]][j + this.mov[k][1]].getNeighbors()[1]) {
                        throw new Exception();
                    }
                    break;
            }
        }
    }
}
```