Tarea 2. Recorrido de una pieza por un tablero NxN

Miruna Andreea Gheata, Rafael Adrián Gil Cañestro

Resumen—En este documento se muestra la implementación de la práctica *Recorregut d'una peça per un tauler de NxN*. Se explicará la estructuración del proyecto, el patrón de programación empleado para el desarrollo, la solución y los distintos problemas encontrados durante el proceso y, finalmente, un manual de usuario.

Index Terms—MVC, Backtracking, Recorrido Euleriano, Java

1. Introducción

En este articulo se detallará por capítulos la implementación de un programa en Java que resuelva el problema del recorrido euleriano dentro de un tablero de NxN.

En el capítulo 2 **se presentará el problema propuesto**, al igual que los requerimientos definidos que se tienen que cumplir.

En el capítulo 3 **se expondrá la solución propuesta**, mencionando a grandes rasgos las técnicas y patrones principales que se han adoptado para resolver el problema: el *patrón MVC*, el *DDD*, la *concurrencia*; y mencionando aspectos importantes de la implementación.

En el capítulo 4 **se especificarán las tecnologías utilizadas**, es decir, el entorno de programación, librerías destacadas, etc.

En el capítulo 5 se explicará la **implementación del Modelo**, separado en Dominio (la definición de los datos) e Infraestructura (las operaciones que se realizan sobre estos datos).

En el capítulo 6 se explicará la **implementación de la Vista**, que se ha separado en varias clases, cada una representando un elemento gráfico.

En el capítulo 7 se explicará la **implementación del Controlador**, explicando los métodos que contiene.

En el capítulo 8 se calculará el **Coste computacional del programa**, al igual que se enseñará cómo varía el tiempo de ejecución dependiendo de la pieza seleccionada y de las dimensiones del tablero.

El capítulo 9 corresponde a una **Guía de usuario** en la que se definen las piezas que se pueden utilizar, los pasos a seguir para poder ejecutar el programa y unos ejemplos de ejecución.

En el capítulo 10 se exponen los **Problemas encontrados** a lo largo de la implementación del programa.

Y, finalmente, el capítulo 11 contiene las **Conclusiones** obtenidas tras realizar este proyecto.

2. Definición del problema

El ejercicio consiste en determinar si una pieza puede recorrer todas las casillas de un tablero de dimensión NxN **visitándolas una única vez**. Este tipo de recorrido recibe el nombre de *Recorrido Euleriano*.

Una pieza está definida por sus movimientos. Por lo tanto, pueden existir varios recorridos que puedan ser una solución correcta del problema, así que se deben mirar todos los posibles movimientos que se puedan hacer para cada posición del tablero, y, además, todos los siguientes movimientos tras realizar un movimiento con el fin de encontrar una solución.

Este problema puede parecer no trivial debido a la gran cantidad de casos que se tienen que tener en cuenta. Sin embargo, existe una técnica de programación que permite comprobar todos estos casos y que, además, siempre encontrará una o varias soluciones si es que existen. Esta técnica se conoce como ***Backtracking***.

3. Solución propuesta

La solución planteada hace uso de distintos patrones, técnicas y formas de programación y de estructuración de proyectos.

En primer lugar, el proceso de cálculo se realizará mediante un *método recursivo* que combinará todos los movimientos posibles, y que usará backtracking en el caso de que se encuentre un movimiento no válido o que no lleve a la solución.

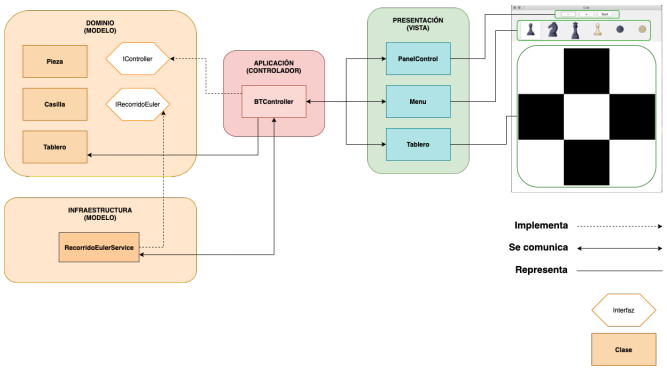


Figura 1: Estructura MVC del proyecto

Backtracking: algoritmo de fuerza bruta que organiza en forma de árbol todas las posibilidades. Recorrerá todos los nodos de ese árbol hasta encontrar una solución, en el caso de que llegase al final y no hubiese encontrado una solución podremos afirmar, sin duda alguna, que no existe solución.

La implementación del programa está hecho siguiendo el conocido ***patrón MVC***.

Finalmente, cabe destacar que se ha implementado la clase encargada de realizar el cálculo como *Thread*, y que además de este existe otro hilo que tiene como función actualizar una barra de progreso durante el proceso de cálculo para que el usuario vea que se está haciendo algo en caso de que el cálculo sea extenso y que no piense que no funciona el programa.

3.1. Patrón MVC y enfoque DDD

Este proyecto esta estructurado basándose en el diseño guiado por el dominio, en adelante DDD, ya que se complementa muy bien con MVC. El DDD no es una tecnología ni una metodología, es un enfoque que proporciona una visión estructural del sistema. DDD separa el *Modelo* en tres partes: ***Dominio, Aplicación e Infraestructura***.

La capa de ***Aplicación*** es responsable de coordinar la ***Infraestructura*** y la capa de ***Dominio*** para hacer una aplicación útil. Por lo general, la capa de *Aplicación* usaría la *Infraestructura* para obtener los datos, consultaría el *Dominio* para ver qué se debería hacer y luego volvería a usar la *Infraestructura* para realizar las operaciones pertinentes para obtener los resultados deseados. [1]. La capa de *Aplicación* por lo tanto se correspondería al *Controlador*, la de *Dominio* e *Infraestructura* al *Modelo*, y la de *Presentación* a la *Vista*.

3.2. Estructuración DDD

A nivel interno el programa está dividido en 4 carpetas (*Presentación*, *Aplicación*, *Dominio*, *Infraestructura*) y un fichero Main.java.

La única función de la clase Main es la de inicializar la *VentanaInicial*, en la que se visualizará el selector de piezas y la dimensión del tablero.

En la carpeta *Presentación* se encuentran todos los elementos gráficos que se usarán. Podemos encontrar:

- las imágenes de las piezas
- la fuente con la que se mostrará en que turno se ha visitado cada casilla y el mensaje de duración del cálculo de la solución
- el ***Tablero*** (de la Vista)
- el ***Menú***
- la ***Ventana***
- el ***Panel de Control***, mediante el cual se puede modificar la dimensión del tablero y poner en marcha el proceso de cálculo de la solución.

En la carpeta *Aplicación* encontramos el ***Controlador*** del programa, el encargado de gestionar la comunicación entre los elementos del proyecto.

En la carpeta *Dominio* encontramos la información sobre los movimientos de cada pieza y las interfaces que implementarán el Controlador (IController) y la Infraestructura (IRecorridoEulerService).

En la carpeta *Infraestructura* esta implementado el algoritmo de backtracking. Como es un `Thread`, encontraremos un método run y el método de la interfaz que es el propio backtracking, recorriendo las casillas hasta encontrar la solución.

3.3. Concurrencia

Se han utilizado `Threads` para el desarrollo de la práctica para poder realizar las operaciones costosas en segundo plano.

La clase `RecorridoEulerService`, que es la que contiene la implementación del algoritmo de backtracking, implementa `Runnable` y dentro de su método de `run()` se realizan los cálculos. En las distintas clases de la Vista se han creado hilos para poder realizar las operaciones necesarias, las llamadas al controlador, las comprobaciones, el pintado de las casillas, etc. Por ejemplo, el siguiente código pertenece al botón de `Start` del *PanelControl* cuando se pone en marcha el programa. La lógica y las consecuencias de pulsar dicho botón se ejecutan en un hilo aparte para no bloquear a la IU.

```
startButton.addActionListener(e -> {
        new Thread(() -> {

        }).start();
});
```

El siguiente código pertenece a la clase *Tablero* (de la Vista) y se puede observar se crea un nuevo hilo dentro del método `paint()` para poder ejecutar el método `pintarCasillasVisitadas()`, ya que este último tiene que ejecutar un doble for y es costoso. Sin embargo, en cuanto se tiene que ejecutar la instrucción de `g2.drawString()` para actualizar la IU se llama al `Thread` principal del Swing IU con la instrucción `SwingUtilities.invokeLater()`. [2] [3]

```
void paint(Graphics g) {
    // Se pinta el orden de visita de las casillas
    new Thread(() -> {
        pintarCasillasVisitadas();
    }).start();
}
void pintarCasillasVisitadas(){
    Graphics2D g2 = (Graphics2D)this.getGraphics();
    for (int i = 0; i < dimension; i++) {
        for (int j = 0; j < dimension; j++) {
            if (controller.isCasillaVisitada(i, j)){
                // Runs inside of the Swing UI thread
                SwingUtilities.invokeLater(() ->
                        g2.drawString(
                        String.valueOf(controller.getOrdenVisitadaCasilla(i, j)),
                        i * lado,
                        j * lado)
                );
            }
        }
    }
}
```

Otro elemento que se ha implementado mediante hilos ha sido la barra de progreso que se actualiza en segundo plano mientras se está ejecutando el cálculo del recorrido.

3.4. Aspectos a destacar

1. Existen dos tipos de Tablero dentro del proyecto: el Tablero del Dominio contiene los datos referidos al tablero, es decir, el número de casillas totales así como también si están visitadas o no, la posición de cada una, la pieza seleccionada por el usuario y la posición inicial elegida para empezar el recorrido. Por otro lado, el Tablero de la Vista se encarga de representar visualmente el Tablero del Dominio. Es importante tener esto claro para evitar confusiones.

2. Se ha intentado mecanizar el proceso de carga de los elementos *Pieza* y la creación de los botones para cada *Pieza*. De esta manera, se ha definido una clase `enum` ***PiezasTablero*** donde se encuentran definidos los nombres de las *Piezas*. Este nombre se corresponde con el nombre de las clases de las *Piezas*, al igual que el nombre de las imagenes que representan dichas *Piezas*.

   ```
   public Tablero(int dimension, int piezaSeleccionada){
       String piezaClass = PiezasTablero.values()[piezaSeleccionada].name();
       String path = "Dominio.Pieza.".concat(piezaClass);
       setClasePieza(piezaPath);
   }
   ```

   Como se puede ver el constructor del Tablero (del Dominio) recibe por parámetro un índice del botón de la *Pieza* seleccionada por el usuario, y con este encuentra el nombre correspondiente de la clase mediante el enumerado *PiezasTablero*.

   El siguiente código (simplificado) es la manera en la que se está creando el objeto de la clase *Pieza* correspondiente.

   ```
   void setClasePieza(String p) {
       Class c = Class.forName(p);
       pieza = (Pieza) c.getDeclaredConstructor().newInstance();
   }
   ```

4. TECNOLOGÍAS UTILIZADAS

4.1. Lenguaje de programación utilizado

La implementación de este proyecto se ha hecho utilizando como lenguaje de programación Java.

4.2. Entorno de programación

La práctica se ha desarrollado en el IDE **IntelliJ IDEA**. **Cabe destacar que si se intenta ejecutar el proyecto el Netbeans no compilará debido a que IntelliJ utiliza carpetas extra (como la de .idea) para poder compilar el proyecto.** Por lo tanto, para ejecutar el proyecto se debe utilizar necesariamente IntelliJ.

Al igual que Netbeans, IntelliJ ofrece una ayuda a la hora de programar elementos gráficos, por lo que se ha facilitado todo el tema del diseño de las ventanas.

4.3. Librerías destacadas

`javax.swing` - se ha utilizado para poder implementar los elementos gráficos: las ventanas, los paneles, los mensajes, los botones, etc.

`java.beans.PropertyChangeListener` y `java.beans.PropertyChangeSupport` - se han utilizado para poder asignar `Listeners` a parámetros para poder saber cuando cambian su valor. Se ha utilizado para poder notificar cuando el proceso de cálculo ha acabado y que el elemento que indica el progreso del cálculo debe finalizar su ejecución también (se explicará con más detalle en los próximos apartados).

5. IMPLEMENTACIÓN DEL MODELO

El *Modelo* es el elemento responsable de mantener la información con la que el sistema opera, estado del tablero, información de las piezas, tanto las consultas como su actualización mediante el algoritmo de backtracking. El *Modelo* también es el encargado de enviarle la información a la *Presentación* (vista), para que esta pinte por pantalla la información necesaria. Por otro lado, las peticiones de actualización o manipulación de los datos llegan al *Modelo* desde el *Controlador*.

Como ya se ha comentado las funciones del *Modelo* están dividas en dos partes. El *Dominio* se encarga mantener, enviar y modificar la información estática del tablero, casillas y piezas. Por el otro lado, la *Infraestructura* es la encargada de realizar las operaciones con los datos, en este caso la implementación del backtracking.

5.1. Dominio

Al ser el encargado de mantener y brindar la información, tendrá que contener la información sobre los elementos gráficos que usará la vista y la información de los elementos que usará la infraestructura para calcular el recorrido.

Dentro del Dominio, por lo tanto, podemos encontrar las definiciones de las distintas estructuras de datos que se usarán para poder representar los elementos del problema:

- La clase *Pieza* contiene la definición de una pieza:

  1. Movimientos de una pieza: definidos con dos arrays uno con los movimientos en el eje x y otro con los movimientos en el eje y,

cuando se seleccionan dos elementos de estos arrays que compartan posición, obtendremos las coordenadas que se tienen que añadir a la posición actual
2. Nombre de la pieza
3. Imagen: se usará para poder representar la pieza en la interfaz de usuario
4. Un booleano que nos dirá si la pieza se ha visto afectada por un redimensionado del tablero (será el caso de la Reina)

Esta clase está definida como `abstract`, y es la clase padre de la que heredan todas las Piezas.

- Las clase hijas de la clase Pieza, y son: *Caballo*, *Dama*, *DamaWhite*, *Peon*, *PeonWhite* y *Reina*

- 2 *interfaces*:

1. `IRecorridoEuler`, esta interfaz solo contiene una función backtrackingEuler, que es la que usará la infraestructura para hacer el cálculo, esta función le entrará por parámetro la posición de la pieza por donde empezará a calcular el recorrido.

2. `IController` esta la usa la *Aplicación*. Las funciones que contiene son las siguientes:

   - `void setInicioPieza(int x, int y)`: coloca la pieza en la posición inicial.
   - `Imagen getImagenPiezaSeleccionada()`: devulve la imagen de la pieza que se ha seleccionado
   - `void startBacktrackingProcess()`: lanza el hilo que calcula el recorrido
   - `void modificarAccesoBotones()`: habilita/deshabilita los botones de las piezas
   - `void modificarAccesoTablero()`: habilita/deshabilita el tablero
   - `void resetearTablero()`: pinta un tablero vacío
   - `void pintarTablero()`: pinta el estado del tablero
   - `void mostrarMensajeAlUsuario (String s)`: vuelve visible un panel con un mensaje para el usuario
   - `boolean isCasillaVisitada(int x, int y)`: comprueba si la posición que le dan pertenece a una casilla visitada.
   - `int getOrdenVisitadaCasilla(int x, int y)`: devuelve la posición en la que la casilla ha sido visitada
   - `void mostrarDuracionEjecucion()`: pintará por pantalla el tiempo que ha tardado en ejecutarse
   - `boolean dentroDeRango(int coordenada)`: comprueba si la coordenada esta dentre del rango del tablero
   - `int cuadrarRangoEnTablero(int coordenada)`: en caso de que se clique fuera del tablero se ajustan las coordenadas

- La clase *Casilla*: tiene un constructor con dos enteros que la ubican en el tablero. También tiene las funciones get de las posiciones x e y, funciones de visita que nos dicen si esta visitada, cambia su estado de visitada y nos dice en que turno se ha visitado la casilla.

- La clase *Tablero*, su constructor consta de un entero que marca el tamaño, un path tipo String que se usa para crear el objeto pieza con la pieza que entra por parámetro. Dentro del mismo constructor se llaman a dos funciones *setClasePieza*, que crea el objeto pieza con el path que le indica la pieza e *inicializarCasillas* que inicializa el tablero creando las casillas y colocándolas en el array bidimensional que compondrá el tablero.

- La clase *Imagen* se encarga de crear los objetos imagen para poder imprimir las piezas por pantalla. En el constructor de la clase podemos ver una imagen contiene además un buffer accesible, que se usa para poder acceder a la imagen deseada mediane el String que nos dan por parámetro. Además, esta clase tiene un único método: **paintComponent(Graphics g, int x, int y, int size)**: se encarga de pintar la imagen en la casilla deseada del tablero.

- La clase *PiezasTablero* solo contiene un enumerado con todos los tipos de piezas.

5.2. Infraestructura

La *Infraestructura* es la encargada de actualizar la información del tablero mediante backtracking. Cuando reciba la pieza y la casilla en la que el usuario quiere que empiece el recorrido, esta tendrá que buscar todos los recorridos que se puedan hacer hasta encontrar un recorrido euleriano. Después le ha de enviar el resultado final a la vista, para que esta lo imprima.

En la carpeta de *Infraestructura* solo encontramos la clase *RecorridoEulerService*, que como la interfaz que implementa, extiende Runnable por lo que el cálculo del backtracking se hará mediante hilos. Cuando se lance el hilo, este inicializara todas las variables de la clase y llamará al método *backtrackingEuler*, donde se encuentra el algoritmo recursivo.

La infraestructura implementa la interfaz IRecorrido-EulerService que tiene los siguientes métodos:

- **start()**: se llama desde el controlador. *Start* lanza el hilo que calculara los recorridos.

- **run()**: este método sobre escrito de la clase Runnable, se ejecuta cuando *start* crea un nuevo hilo y lo lanza.

Este método inicializa las variables que usará el algoritmo de backtracking que són:

- **boolean hayRecorrido** será false, si se encuentra un recorrido se cambia a true.
- **int casillasRecorridas** será 1, ya la casilla inicial se cuenta
- **int tamaño** será el cuadrado de la dimensión que es el lado, se usa para saber si se han visitado todas las casillas
- **long duraciónEjecucion** será igual al tiempo transcurrido hasta el inicio del backtracking.

- **backtrackingEuler(int x, int y)**:Va visitando todas las casillas a las que se puede mover la pieza, cuando el algoritmo visita una nueva casilla, realiza una llamada recursiva. En el caso de que se tenga que retroceder cambia el estado de la casilla a no visitado, decrementa el contador de visitas y termina la ejecución, volviendo a la casilla anterior. Este algoritmo **siempre encontrará un recorrido**, no siempre el más óptimo, y en caso de que no lo encuentre, podremos afirmar que no existe.

```
void backtrackingEuler(int x, int y){
int xt, yt;
    if (casillasRecorridas != tama o) {
        for (int i = 0; i < tablero.getPieza().getNumMovs(); i++) {
            xt = x + tablero.getPieza().getMovX(i);
            yt = y + tablero.getPieza().getMovY(i);
            if (isMovimientoValido(tablero, xt, yt)){
                if (!tablero.getCasilla(xt, yt).isVisitada()){
                    casillasRecorridas++;
                    tablero.getCasilla(xt, yt).setVisitada(true, casillasRecorridas);
                    backtrackingEuler(xt, yt);
                    if(casillasRecorridas != tama o){
                        casillasRecorridas--;
                        tablero.getCasilla(xt, yt).setVisitada(false, -1);
                    } else {
                        hayRecorrido = true;
                        return;
                    }
                }
            }
        }
    }
}
```

- **isMovimientoValido(Tablero tablero, int x, int y)**: comprueba que el movimiento que se quiere hacer está dentro de los límites del tablero

- **getDuracionEjecucion()**: devulve la duración de la ejecución del algoritmo de backtracking.

6. Implementación de la Vista

La *Vista* es la responsable de representar gráficamente el modelo. Como mínimo, tiene que ser capaz de representar el número de piezas del que se dispone, el tablero y algún control de las dimensiones de dicho tablero. Además, el usuario tiene que poder colocar una pieza libremente en el tablero. Con el fin de llevar esto a cabo, se ha dividido la representación de cada requisito en distintas clases:

1. ***VentanaInicial***: Es la ventana inicial que aparece cuando se levanta el programa [2]. Contiene:
   - un selector de piezas
   - un selector de las dimensiones del tablero. **El rango de la dimensión varia desde 3x3 a 8x8**.

2. ***Ventana***: contiene el *Menu*, el *Tablero*, el *PanelControl* y un *JOptionPane* que sirve para poder mostrar mensajes al usuario (este último esta definido dentro de la misma clase Ventana, no hay un elemento aparte) [3]. Sirve simplemente como contenedor.

3. ***Menu***: Es el selector de pieza que se puede ver en la figura [3].

4. ***Tablero***: El tablero NxN sobre el cual se tiene que encontrar el recorrido euleriano. Este tablero es interactivo, de manera que el usuario puede poner piezas en las casillas. Una vez acabado el programa, y en caso de que se encuentre una solución, aparecerá dentro de cada casilla un número que se corresponde con el orden en el que se han visitado [4].

5. ***PanelControl***: este panel sirve como elemento de control con el que el usuario puede interactuar con el programa. Tiene 4 elementos:
   - **Botones para modificar el tamaño del tablero**. Una vez pasado de la VentanaInicial y elegida la dimensión, el usuario puede aún modificar las dimensiones como desee. Además, si hay ya una pieza colocada en el tablero, la pieza se irá recolocando por el tablero manteniendo la posición anterior. Por ejemplo, si la pieza estaba colocada en la última casilla de la primera fila y se disminuye el tamaño, la pieza seguirá estando en la última casilla de la primera fila.
   - **Botón de control de la ejecución**. Este botón sirve para que el usuario pueda iniciar el cálculo del recorrido y para resetear el tablero una vez realizado un recorrido.
   - **Mensaje del tiempo total de la ejecución**: una vez acabado el cálculo, aparecerá un mensaje con la duracion total del cálculo. Este aparece en **nanosegundos**.
   - **Barra de progreso**: esta barra aparece durante el proceso de cálculo y sirve para que el usuario pueda ver que se está ejecutando el programa. En los casos en el que el cálculo es muy largo, esta barra servirá para ver que no se ha quedado bloqueado el programa, sino que simplemente aún no se ha llegado a la solución. Esta barra se va rellenando y reseteando cada vez que se llena.

   Existe un 5to elemento, el *JOptionPane*, aunque este no se ha definido en una clase aparte sino que está dentro de la misma clase *Ventana*. Este sirve para poder mostras 3 tipos de mensaje al usuario:
   - Mensaje de error cuando no se ha colocado una pieza en el tablero y se quiere iniciar el cálculo
   - Mensaje de solución encontrada

- Mensaje de solución no encontrada

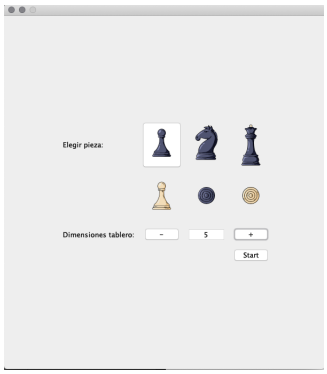


Figura 2: Ventana Inicial

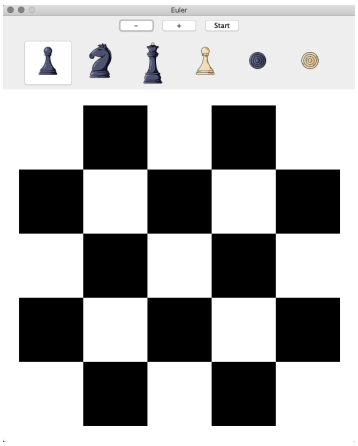


Figura 3: Ventana con los elementos PanelControl, Menu y Tablero

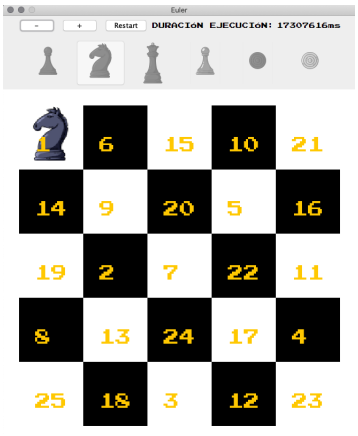


Figura 4: Tablero con el orden de visita de la pieza cuando hay solución

7. Implementación del Controlador

El *Controlador* es el elemento que monitoriza y controla las comunicaciones entre la Vista y el Modelo. Para facilitar la comprensión del lector se llamará al Tablero referente al de la Vista ***tableroPresentacion***, y al tablero del Dominio (que contiene los datos) ***tableroDominio***.

El controlador implementa la interfaz `IController` que tiene los siguientes métodos:

`void setInicioPieza(int x, int y)`: se llama desde el *tableroPresentacion* cuando el usuario coloca una pieza en el tablero. El Controlador envía éstas coordenadas al *tableroDominio* y la guarda dentro de una variable `Casilla inicioPieza`.

`Imagen getImagenPiezaSeleccionada()`: se llama desde el *tableroPresentacion* para poder obtener la imagen de la Pieza seleccionada por el usuario del *tableroDominio*.

`void startBacktrackingProcess()`: se llama desde *PanelControl* cuando el usuario pulsa el botón de `Start` y hay una pieza colocada en el casilla. Esta método llama internamente al método `start()` de la clase ***RecorridoEulerService*** (a la Infraestructura) para crear el `Thread` que ejecutará el cálculo.

`void modificarAccesoBotones()`: se llama desde *PanelControl* para habilitar o deshabilitar los botones de las piezas del *Menu* cuando:

- deshabilitar cuando se pone en marcha el proceso de cálculo del recorrido, por lo tanto el usuario no debe tener la posibilidad de modificar la pieza seleccionada
- habilitar cuando el cálculo ha acabado y el usuario le ha dado al botón de `Resetear`, por lo tanto debe tener acceso a las piezas

`void modificarAccesoTablero()`: al igual que el método anterior, se llama desde *PanelControl* y sirve para habilitar o deshabilitar el *tableroPresentacion* cuando:

- deshabilitar cuando se pone en marcha el proceso de cálculo del recorrido, por lo tanto el usuario no debe tener la posibilidad de modificar la casilla inicial
- habilitar cuando el cálculo ha acabado y el usuario le ha dado al botón de `Resetear`, por lo tanto debe tener la posibilidad de poner una pieza en el *tableroPresentacion*.

`void modificarDimensionesTablero(int nuevasDimensiones, boolean decrementar)`: se llama desde el *PanelControl* cuando el usuario modifica las dimensiones del tablero (las aumenta o las disminuye). Esta llamada internamente hace las siguientes operaciones:

1. Actualiza las dimensiones del *tableroDominio* a las nuevas dimensiones deseadas
2. Comprueba si está definida la casilla inicial de la Pieza:
   - Si está definida y el usuario ha disminuido las dimensiones (se sabe esto mirando el parámetro `decrementar`), se mira si

las coordenadas están en el límite del *tableroDominio* y si lo están se modifica el inicioPieza (disminuyendo las coordenadas en una unidad).
En caso contrario, no se actualizan los valores de la `Casilla inicioPieza`, pero sí que se hace el `setInicioPieza(inicioPieza.X, inicioPieza.Y)` debido a que al resetear las dimensiones del *tableroDominio* se han reseteado todas las Casillas y se ha perdido la posición seleccionada.

3. Actualiza las dimensiones del *tableroPresentacion* llamando al tableroPresentacion.actualizarDimensiones(nuevasDimensiones), método donde se recalcula el lado de las `Casillas` del y hace el repaint() del *tableroPresentacion*.

`void mostrarCaminoEuler(boolean hayRecorrido)`: se llama cuando el proceso de cálculo ha finalizado. El `RecorridoEulerService` llama a este método para que el controlador haga todas las operaciones necesarias para mostrar el final de la ejecución y la obtención de una solución. Dentro de este método:

1. Se avisa al proceso que actualiza la barra de progreso que no hace falta seguir
2. envia el tiempo total de ejecución al *PanelControl* mediante `mostrarDuracionEjecucion()`
3. habilita el acceso de los botones del *Menu* (los botones de las piezas)
4. en caso de que haya solución, fuerza el `repaint()` del *tableroPresentacion* para que aparezcan en las casillas el orden de visita [4]
5. por último, muestra un mensaje de confirmación por pantalla mediante el `mostrarMensajeAlUsuario()` en el que se avisará si se ha encontrado o no una solución

`void resetearTablero()`: método que "limpia" el *tableroPresentacion* y el *tableroDominio*. Para el *tableroPresentacion*, se quita la pieza del tablero, y para el *tableroDominio* se resetean todas las casillas (es decir, se ponen que no están visitadas y se pone el orden de visita a -1) y se resetea el `inicioPieza` a (-1, -1). Se llama desde dos sitios:

- Desde el método `setPiezaSeleccionada` del propio controlador cuando había una pieza ya seleccionada y colocada en el *tableroPresentacion* pero el usuario quiere cambiar de pieza.

- Desde el *tableroPresentacion* cuando el usuario pulsa el botón de `Resetear` del *PanelControl*.

`void pintarTablero()`: se llama cuando el proceso de cálculo se ha acabado y que quiere repintar el *tableroPresentacion* para que se muestren las casillas con el orden de visita actualizada en caso de que haya solución. Este método llama al tableroPresentacion.repaint(). El que lo llama es el propio controlador desde el método `mostrarRecorridoEuler`.

`void mostrarMensajeAlUsuario(String s)`: se llama:

- al acabar el cálculo y sirve para notificar al usuario de si se ha conseguido encontrar una solución o no (se llama desde el método `mostrarRecorridoEuler`)

- desde el *PanelControl* para notificar al usuario de que no se ha colocado una pieza en el tablero y que no puede empezar la ejecución del programa hasta que no lo hace

`boolean isCasillaVisitada(int x, int y)`: sirve para acceder a la casilla que se encuentra en la posición (x, y) del *tableroDominio* y ver si se ha visitado ya o no. Se llama desde el *tableroPresentacion* cuando se están pintando las casillas visitadas.

`int getOrdenVisitadaCasilla(int x, int y)`: sirve para acceder a la casilla que se encuentra en la posición (x, y) del *tableroDominio* y obtener el número de visita. Se llama desde el *tableroPresentacion* cuando se están pintando las casillas visitadas.

`void mostrarDuracionEjecucion()`: se llama desde el propio controlador cuando se ha acabado el cálculo para poder mostrar por el *PanelControl* la duración total de la ejecución.

`boolean dentroDeRango(int coordenada)`: comprueba que la coordenada pasada por parámetro está dentro del rango del *tableroDominio*. Se llama desde el *tableroPresentacion* cuando el usuario ha colocado una pieza dentro del *tableroPresentacion*.

`int cuadrarRangoEnTablero(int coordenada)`: llama al *tableroDominio* para que cuadre la coordenada dentro de los rangos y devuelve el valor correcto. Se llama desde el *tableroPresentacion* cuando el usuario ha colocado una pieza dentro del *tableroPresentacion*.

8. COSTE COMPUTACIONAL

El análisis del tiempo de ejecución de un programa recursivo vendrá en función del tiempo requerido por la(s) llamada(s) recursiva(s) que aparezcan en él.

Para este algoritmo recursivos en concreto, debido a que decrece a velocidad constante (cada vez se llama al método recursivo con 1 movimiento), se debe calcular el coste mediante lo siguiente:

$$T_1(n) \in \begin{cases} O(n^k) & , si\ a < 1 \\ O(n^{k+1}) & , si\ a = 1 \\ O(a^{n/b}) & , si\ a > 1 \end{cases}$$

Recordemos cuáles eran las parámetros que se necesitan:

- a: número máximo de llamadas recursivas internas por cada llamada externa
- b: constante del cambio
- n: número iteraciones
- k: máximo de hacer todos los casos base o de una composición

El coste de realizar backtracking suele ser factorial o exponencial. Para el algoritmo de backtracking desarrollado en la clase `RecorridoEulerService`:

- a: m, siendo m el número total de movimientos, ya que se realizan tantas llamadas como movimientos tenga la pieza
- b: 1, porque cada llamada sólo se hace con un único movimiento
- n: m, ya que el se comprueban todos los movimientos que puede realizar la pieza
- k: 1, porque la suma de hacer todos los casos base es m y para que la siguiente igualdad sea cierta $O(n^k) = n$, k tiene que ser 1

Por lo tanto, estamos en el tercer casi, así que tenemos que el coste computacional de este algoritmo es

$$O(a^{n/b}) = O(n^{n/1}) = \boldsymbol{O(n^n)}$$

8.1. Incremento del tiempo de ejecución según las dimensiones del tablero

La duración de la ejecución tiene una **alta correlación con la dimensión del tablero y la pieza seleccionada**. Las distintas piezas tienen sus propios movimientos y estos pueden ser mejores o peores. Otras tienen más o menos movimientos, por lo que el número de llamadas recursivas que se deben realizar para encontrar una solución son diferentes.

De esta manera, se puede observar en la figura [5] la evolución de la duración (en nanosegundos[1]) para cada pieza a medida que se van aumentando las dimensiones del tablero. Cabe destacar que las piezas que más tiempo tardan son el **Caballo** [5b], la **Dama** [5e] y la **Dama Blanca** [5f].

9. GUÍA DE USUARIO

9.1. Definición de las piezas

Se han definido 6 piezas distintas: 3 de ellas realizan los mismos movimientos definidos en el juego de Ajedrez (la Reina, el Caballo y el Peón), y las otras 3 son definidas con nuevos movimientos.

Reina: Se puede mover en todas las direcciones hasta n casillas en cada movimiento. [6a]

Caballo: Se mueve "haciendo Ls"por el tablero. [6b]

Peón: Sólo se puede mover 1 casilla a la vez hacia abajo. [6c]

Peón blanco: Se puede mover 1 casilla [6d]:

- hacia abajo
- a la derecha
- a la izquierda

Dama: Hace los mismos movimientos que la Reina pero sólo se puede mover 1 casilla cada vez. Por lo tanto, se puede mover [6e]:

- hacia abajo
- hacia arriba
- a la izquierda
- a la derecha
- en las diagonales

Dama blanca: Hace los mismos movimientos que la Dama, pero cambia el número de casillas que se puede mover [6f]:

- 1 casilla: hacia arriba, hacia abajo, a la derecha y a la izquierda
- 2 casillas: en las diagonales

9.2. Ejecución del programa

Para poder ejecutar el programa se debe:

1. Abrir el proyecto con IntelliJ IDEA
2. Compilar el proyecto
3. Ejecutar el programa

Una vez hecho esto aparecerá la primera ventana del programa, en la que se podrá elegir la pieza deseada y poner el tamaño que se quiere que tenga el tablero. El tablero puede tener una dimensión de 3x3 hasta 8x8.

Por defecto el tablero tendrá una dimensión de **5x5** y la pieza seleccionada será el **Peón**. Una vez pulsado el botón de `Start`, saldrá una ventana que contiene un tablero, un menú con las piezas y unos botones de control.

Si se desea modificar el tamaño del tablero, pulse el botón de '-', y si lo quiere hacer más grande pulse '+'.

Para poder colocar una pieza en el tablero simplemente debe seleccionar una de las casillas del mismo. Por defecto la pieza seleccionada es el **Peón**, pero si se desea cambiar de pieza se debe seleccionar desde el menú de arriba.

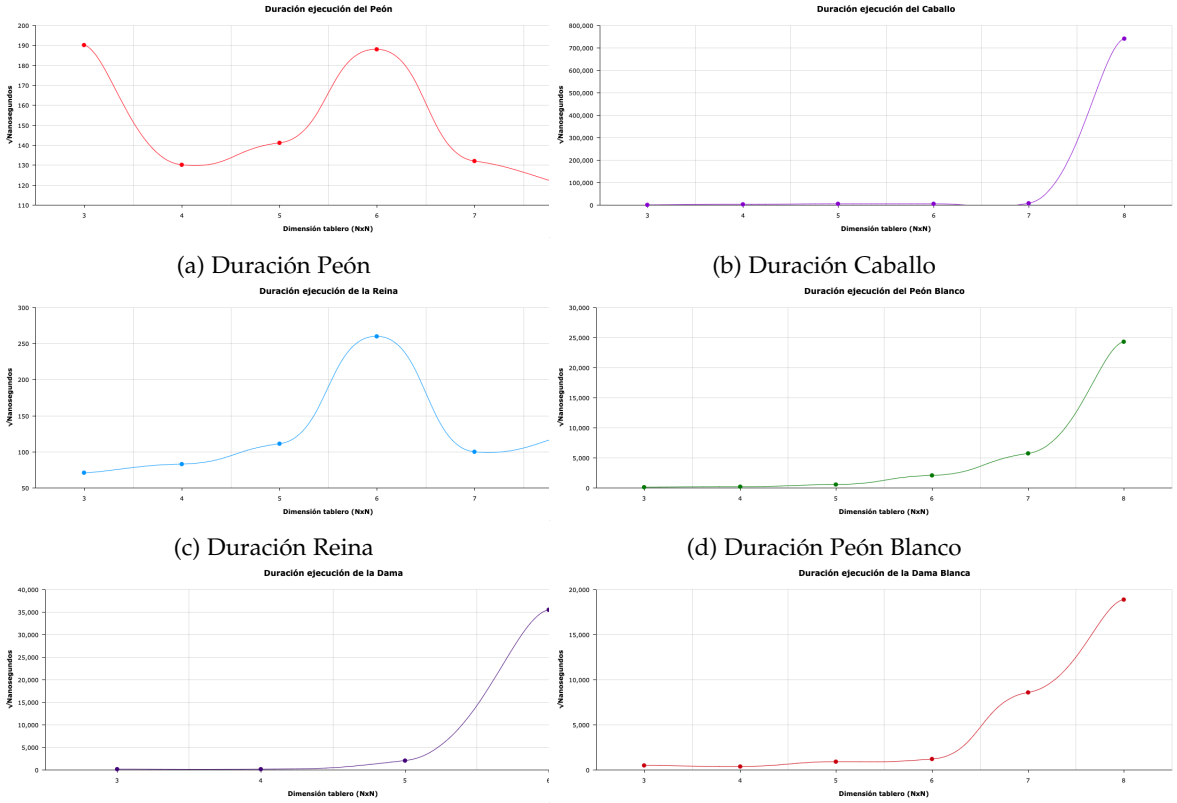
Una vez colocada la pieza en la posición inicial deseada, simplemente se debe pulsar el botón de `Start` para empezar a buscar un recorrido sobre este tablero. En caso de que la pieza no esté colocada en el tablero saltará un mensaje de error. [8]

La duración del proceso de cálculo varía dependiendo de

- la pieza seleccionada
- el tamaño del tablero

y en caso de que sea más lento se puede ver una barra de progreso que se está actualizando constantemente para poder mostrar que el proceso aún está calculando.

1. Se ha hecho la raíz cuadrada del tiempo total de ejecución



(a) Duración Peón

(b) Duración Caballo

(c) Duración Reina

(d) Duración Peón Blanco

(e) Duración Dama

(f) Duración Dama Blanca

Figura 5: Duración de la ejecución por pieza a medida que se incrementa la dimensión del tablero

Una vez finalizado aparecerá por pantalla un mensaje de confirmación, informando si se ha encontrado una solución [7] o no. En caso de que haya aparecerá en cada casilla en orden en el que ha sido visitado.

Para poder volver a ejecutar el programa se debe pulsar el botón de `Resetear`, que limpiará el tablero y así se podrá volver a intentarlo desde una nueva posición inicial o con una pieza diferente.

9.3. Ejemplos de ejecuciones

En la figura [7] se pueden observar los pasos realizados para encontrar una solución con el Caballo.

En la figura [8] se puede observar qué ocurre en caso de que no haya ninguna pieza colocada en el tablero pero se quiere empezar el programa.

10. Problemas encontrados

10.1. Implementación del algoritmo de backtracking

El algoritmo se implementó inicialmente de forma recursiva y pintando cada vez la pieza cuando se realizaba un movimiento. Este hecho hacía que el proceso sea mucho más lento. Sin embargo, se quería conservar el pintado del movimiento para poder ofrecer una manera de visualizar como se realizan los movimientos.

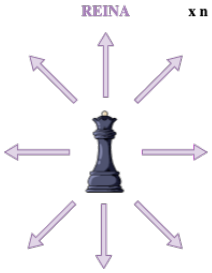
Una vez añadidas las visitas a las casillas y viendo la ejecución que ahora, además de tardar muchísimo, era muy confuso ver como las piezas se pintaban y cambiaban de orden de visita debido al backtracking, se eligió pintar el tablero al final del proceso de backtracking, mostrando de esta manera la solución final y tardando muy poco tiempo en comparación con el algoritmo anterior.

También se había planteado realizar la solución de forma iterativa en vez de recursiva mediante el uso de pilas. El algoritmo era considerablemente más rápido que antes, pero el hecho de tener que conservar el estado anterior dentro de las pilas (es decir, el movimiento anterior), y saber exactamente cómo seguir el flujo correcto para poder guardar y realizar todos los movimientos complicaba mucho la implementación.

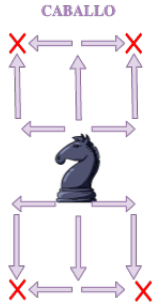
Por lo tanto, al final se ha decidido que realizar el algoritmo recursivo era mucho más eficiente, más sencillo y ofrecía unos mejores resultados.

11. Conclusión

En esta práctica hemos podido ver como usar MVC facilita mucho el trabajo, brindándonos una visión más clara del proyecto. Además, resulta más fácil añadir mejoras y corregir errores en el funcionamiento durante el desarrollo cuando el programa no hace lo que tu esperabas gracias a la división de este patrón.

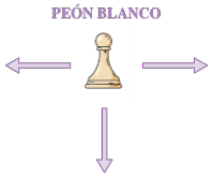


(a) Movimiento reina

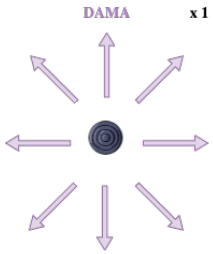


(b) Movimiento Caballo

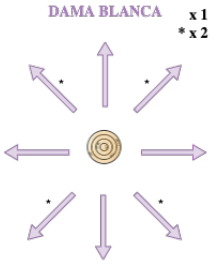
(c) Movimiento Peón



(d) Movimiento Peón Blanco



(e) Movimiento Dama



(f) Movimiento Dama Blanca

Figura 6: Movimientos de las piezas

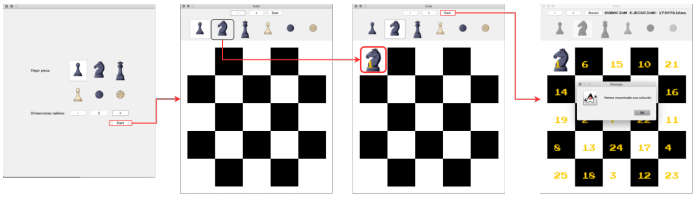


Figura 7: Flujo de la ejecución del programa cuando se encuentra una solución

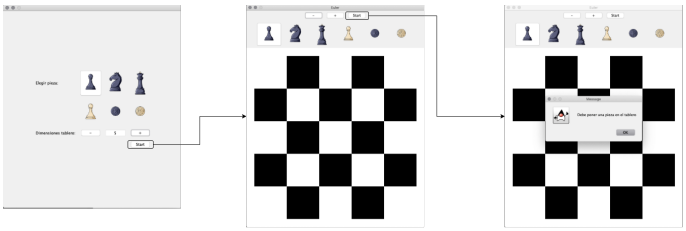


Figura 8: Ejecutar el programa sin colocar una pieza en el tablero

REFERENCIAS

[1] FEDERICO *Domain-Driven Desing and MVC Architectures* https://blog.fedecarg.com/2009/03/11/domain-driven-design-and-mvc-architectures/

[2] *How to Use Threads in Java Swing* https://stackabuse.com/how-to-use-threads-in-java-swing/

[3] *Threading with Swing: SwingUtilities.invokeLater()* https://www.javamex.com/tutorials/threads/invokelater.shtml