Backtracking

Martin Mitrovski Delov, Josep Antoni Naranjo Llompart, Sergi Moreno Pérez, Pau Rosado Muñoz

Resumen — El ejercicio consiste en una variante del recorrido de una pieza de ajedrez por un tablero de NxN. El programa solución debe de presentar una estructura de diseño Modelo Vista Controlador (MVC) y se debe utilizar *backtracking* para manejar las piezas.

Por motivos de tamaño del fichero de la práctica, hemos decidido subir el vídeo a YouTube y no incluirlo en el archivo comprimido entregado. El enlace al vídeo de la práctica en YouTube es el siguiente:
https://www.youtube.com/watch?v=i3xpGfiGOUU

I. INTRODUCCIÓN

En este artículo se explicará cómo se ha realizado la segunda práctica de la asignatura Algoritmos Avanzados. Se desarrollarán algunos conceptos teóricos expuestos en el segundo tema y también se describirá la implementación de una aplicación que mostrará un tablero de ajedrez de tamaño NxN en el que podremos colocar piezas de hasta 6 tipos definidos. El algoritmo realizará un recorrido de casilla única del tablero, este es aquél donde la pieza visita todas las casillas, pasando una sola vez por cada una de ellas.

Para los conceptos del segundo tema, trataremos: el entorno de programación utilizado, los conceptos de MVC, el patrón por eventos, que es el *backtracking*, la recursividad, como afecta el *backtracking* a la memoria, su esquema general, cuando usarlo, coste, el concepto de NP-Completo y dos de los problemas clásicos del backtracking: problema de las N-Reinas y el problema de la mochila. Finalmente haremos una descripción de la aplicación en la que veremos el funcionamiento y la implementación de todo lo aprendido.

Por parte del concepto MVC, explicaremos qué es y por qué es el método de diseño que utilizaremos durante todo el curso para la implementación de las prácticas. Explicaremos como este método divide nuestra práctica en diferentes partes y su implementación específica, y como hace que los componentes del MVC se comunican.

Después se definirá que es el *backtracking*, cuál es su esquema general y el análisis del coste y cuando deberíamos usar este tipo de algoritmos.

Una vez explicadas estas secciones teóricas, procederemos a mostrar la implementación de la aplicación usando el patrón de diseño MVC y explicando cada una de las partes, junto a los conceptos de código más importantes. Haremos bastante hincapié en el nivel de abstracción que se ha perseguido obtener para la implementación y comunicación entre los diferentes componentes de la arquitectura. Primero explicaremos el modelo de datos, después el controlador y finalmente la vista o interfaz gráfica.

Del apartado gráfico, explicaremos algunas características para asegurar la consistencia del programa.

Para acabar habrá un apartado de conclusiones dónde se realizará una breve reflexión sobre los resultados obtenidos y sobre el trabajo dedicado al proyecto.

II. ENTORNO DE PROGRAMACIÓN

En nuestra práctica usaremos el entorno de programación NetBeans, entorno de desarrollo integrado libre, orientado principalmente al desarrollo de aplicaciones Java. La plataforma NetBeans permite el desarrollo de aplicaciones estructuradas mediante un conjunto de componentes denominados “módulos”. La construcción de aplicaciones a partir de módulos las permite ser extendidas agregándoles nuevos módulos. Como los módulos pueden ser desarrollados independientemente, las aplicaciones implementadas en NetBeans pueden ser extendidas fácilmente por otros desarrolladores de *software*.

III. MODELO VISTA CONTROLADOR (MVC)

El modelo-vista-controlador (MVC) [1] es una arquitectura de software que separa los datos, la interfaz de usuario y la lógica de control en tres componentes distintos. El modelo representa los datos, la vista muestra los datos al usuario de forma visualmente atractiva y el controlador actúa como intermediario entre el modelo y la vista, gestionando el flujo de información entre ellos. El flujo de una aplicación que usa el MVC comienza con la interacción del usuario con la interfaz de usuario, el controlador recibe la acción del usuario y actualiza el modelo acorde a la acción, luego la vista utiliza los datos del modelo para generar la interfaz adecuada para el usuario.

Existen tres formas básicas de implementación del MVC: distribuido, centralizado y centralizado en el controlador. El uso de este patrón de diseño por capas tiene varias ventajas, como la división del trabajo, la reutilización de código, la

flexibilidad y escalabilidad que supone, así como la facilidad de realizar el *testing* de la aplicación. Sin embargo, también tiene algunas desventajas, como la complejidad adicional que puede presentar debido a que requiere más código y esfuerzo para mantener la separación de tareas entre los componentes.

En la Figura 1 aparece el esquema habitual del patrón de diseño MVC.

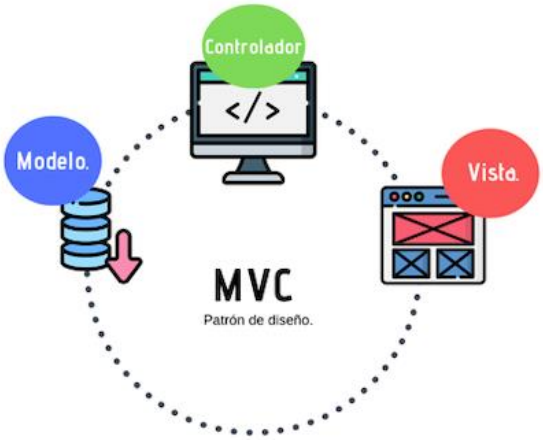

Figura 1: Modelo Vista Controlador

IV. PATRÓN POR EVENTOS

La arquitectura orientada a eventos o *Event-driven architecture* (EDA) es una forma de diseñar software que se enfoca en los eventos que ocurren dentro del sistema [2]. Los eventos son ocurrencias notificadas, como cuando un usuario hace clic en un botón. En una arquitectura EDA, los componentes del sistema están diseñados para reaccionar a estos eventos en tiempo real. Los componentes se comunican entre sí mediante eventos, propagándose a los componentes relevantes que responderán al evento adecuadamente.

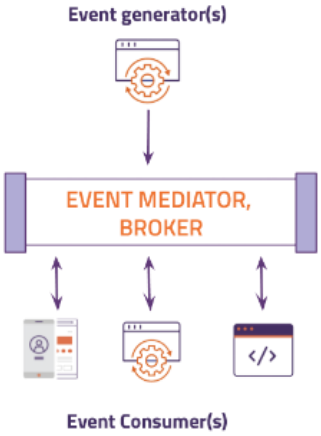

Figura 2: Patrón por eventos

La arquitectura de eventos es especialmente útil [3] en sistemas que requieren procesamiento en tiempo real, escalabilidad y flexibilidad, como redes de sensores, comercio financiero, sitios web de comercio electrónico y redes sociales. Los sistemas basados en esta arquitectura pueden reaccionar rápidamente a cambios y son tolerantes a fallos debido al bajo acoplamiento entre los componentes.

Sin embargo, esta arquitectura también puede tener desventajas, como la complejidad y dificultad de depuración debido a la naturaleza distribuida del sistema y la necesidad de sincronización de los eventos. También se deben tener en cuenta posibles problemas de latencia, ya que los eventos pueden tardar un cierto tiempo en propagarse a través de los componentes.

V. *BACKTRACKING*

El *Backtracking* [4] es una técnica de resolución de problemas que se utiliza en la programación en la cual buscamos todas las posibles soluciones a un problema mediante la exploración de todas las posibles soluciones.

Esta técnica implica la búsqueda de soluciones de manera recursiva, en la que se prueba cada opción posible para llegar a la solución. Si se encuentra una solución que no funciona, se retrocede (*backtrack*) y se prueba otra opción hasta que se encuentra una solución viable o se comprueba que no existe ninguna solución. La figura 3 ilustra esta explicación.

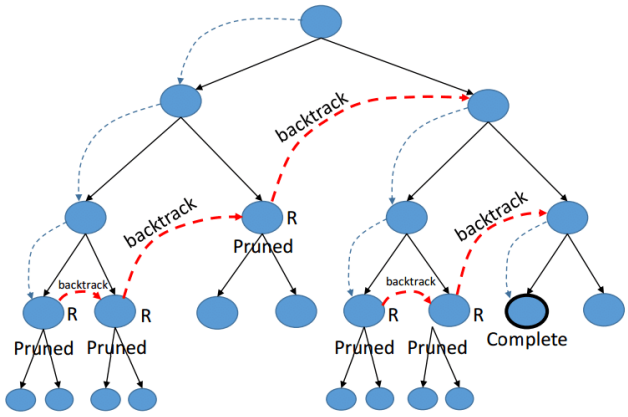

Figura 3: Backtracking

Se trata de un tipo de solución más fácil e intuitiva ya que sigue un proceso similar al que un ser humano seguiría para resolver un problema de forma manual.

Un ejemplo de esto sería solventar un problema de *sudoku* con *backtracking*. El proceso consistiría en asignar un número a una celda, comprobar si esa asignación es válida y seguiríamos asignando número hasta encontrar una solución o

detectar un error. Este tipo de solución es el que nosotros las personas usuraríamos para resolver el Sudoku de forma manual lo que puedes hacer que el enfoque sea más intuitivo.

Pero es importante saber que puede no ser la mejor opción en todos los casos y que a menudo existen algoritmos más eficientes para resolver dichos problemas.

En resumidas cuentas, el *Backtracking* tiene algunas ventajas, como la capacidad de encontrar soluciones en problemas de alta complejidad de una manera sistemática y organizada. Sin embargo, también tiene limitaciones, como el hecho de que puede ser computacionalmente costoso, especialmente para problemas muy grandes, y que puede requerir mucho tiempo y esfuerzo para desarrollar algoritmos eficientes.

V.I MEMORIA RAM, MEMORIA HEAP Y MEMORIA PILA

Los algoritmos de *backtracking* pueden consumir una cantidad significativa de memoria RAM y pueden afectar tanto al *heap* como a la pila de memoria.

Como hemos explicado previamente el *backtracking* implica explorar diferentes ramas de una solución potencial, así como aparece representado en la Figura 3, lo que significa que el algoritmo debe almacenar la información de cada ramificación para poder volver a ella más tarde (*Backtrack*). Si el árbol de soluciones es muy grande, esto puede resultar en una cantidad significativa de almacenamiento en memoria.

Para el caso de la memoria del *heap*, los algoritmos de *backtracking* pueden crear objetos y estructuras de datos en memoria para representar diferentes soluciones potenciales, lo que puede aumentar el tamaño del *heap*. A medida que llamamos un procedimiento en el *heap*, todo esto se va guardando y cuando retornan se libera esta memoria. Su función es un símil a un muelle que sube y baja. Si se utiliza una estructura de datos inadecuada para representar las soluciones potenciales, se puede desperdiciar espacio en el *heap*.

En cuanto a la memoria de la pila, el *backtracking* puede utilizar una cantidad significativa de memoria de la pila debido a la recursividad. Cada vez que se llama a una función recursiva, se añade una nueva entrada en la pila para almacenar la información de la llamada a la función, y esto puede acumularse si hay muchas llamadas recursivas.

Es importante tener en cuenta el uso de la memoria por los algoritmos de *backtracking*. Por lo cual se debe evaluar cuidadosamente el uso de la memoria en cada caso y optimizar el algoritmo para minimizar la cantidad de memoria utilizada.

V.II RECURSIVIDAD

La recursividad [5] se refiere a una función o proceso que se llama a sí misma repetidamente en su ejecución. Es una técnica común de la programación la cual se utiliza para resolver problemas que pueden ser descompuestos en problemas más pequeños (subproblema) con lo cual, en lugar de abordar el problema original directamente, resolvemos cada subproblema recursivamente.

Es necesario establecer un caso base que permita a la función finalizar la recursión y evitar caer en una llamada infinita.

Un ejemplo fácil para entender mejor la recursividad sería el cálculo factorial de un número entero.

La factorial de un número entero n (se denota como: n!), es el producto de todos los números enteros positivos desde 1 hasta n. Es decir:

n! = 1 x 2 x 3 x … x n

Por ejemplo, si n = 5, su factorial sería:

5! = 1 x 2 x 3 x 4 x 5 = 120

Para calcular la factorial de 5 utilizando recursividad, podemos definir una función recursiva que se llame a sí misma para calcular la factorial de n-1. La función tendría un caso base para cuando n=1, en cuyo caso la factorial sería simplemente 1.

Si llamamos a esta función con un argumento de 5, se llamará a sí misma con un argumento de 4, luego con 3, luego con 2, y finalmente con 1 (el caso base). La función comenzará a regresar valores una vez que alcance el caso base, y se multiplicará cada valor de retorno por n para obtener el factorial completo. La figura 4 ilustra todo esto explicado.

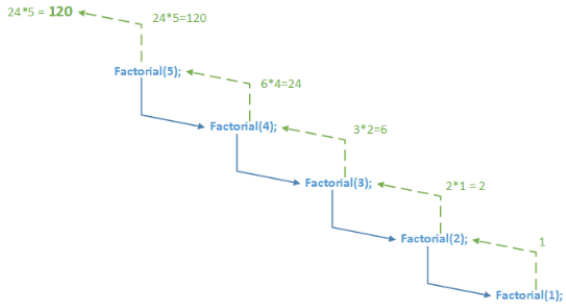

Figura 4: Cálculo de la factorial de un número entero

V.III ESQUEMA GENERAL

Consideremos una solución al conjunto (X_1, X_2, …, X_n).

El esquema general del *backtracking* se puede describir en los siguientes pasos:

1. Definir el problema y su objetivo: Es necesario comprender el problema y definir claramente cuál es el objetivo que se desea alcanzar.

2. Representar el problema en términos de un árbol de soluciones: El árbol de soluciones es una representación gráfica de todas las posibles soluciones al problema. (Figura 3)

3. Definir las condiciones iniciales: Las restricciones y los valores iniciales que se utilizarán para explorar las soluciones.

4. Definir la función de selección de opciones: La función de selección de opciones es la que determina cuál será la próxima opción para explorar.

   En cada instante el algoritmo se encuentra en un estado k que es solución parcial (X_1, X_2, …, X_k). Siendo k < n. En este punto:

   a. Si se puede añadir un nuevo elemento y avanzar hasta el estado parcial de solución k+1, se hace.
   b. Si no, se prueba con otros elementos a añadir como representantes de parte de la solución X_{k+1}.
   c. Si no quedan más elementos por probar, se deshace el camino y se retorna al estado k-1.
   d. Esto se repite recursivamente hasta que la solución parcial es una solución válida del problema que se quiere resolver, o hasta que no quedan más soluciones parciales por analizar, en cuyo caso podemos afirmar que el problema no tiene solución.

5. Definir la función de validación de opciones: La función de validación de opciones es la que comprueba si la opción seleccionada es válida y cumple con las restricciones del problema.

6. Definir la función de solución: La función de solución es la que comprueba si se ha alcanzado la solución deseada.

7. Si la función de solución no se cumple, seleccionar una nueva opción y validarla: Si la función de solución no se cumple, se debe seleccionar una nueva opción y validarla utilizando la función de selección de opciones y la función de validación de opciones.

8. Si la opción seleccionada no es válida, volver a seleccionar otra opción: Si la opción seleccionada no es válida, se debe volver a seleccionar otra opción y validarla.

9. Repetir los pasos 7 y 8 hasta que se encuentre una solución o se hayan explorado todas las opciones posibles.

10. Si se encuentra una solución, mostrarla y volver al paso 7 para buscar más soluciones.

11. Si no se encuentran más soluciones, el proceso termina.

En resumen, el esquema general del backtracking consiste en definir el problema, representarlo en términos de un árbol de soluciones, definir las condiciones iniciales, seleccionar opciones, validarlas, buscar soluciones y repetir el proceso hasta encontrar todas las soluciones posibles.

V.IV CUANDO USAR EL *BACKTRACKING*

El *backtracking* es una técnica poderosa que se puede utilizar en una amplia variedad de problemas en los que se desea explorar sistemáticamente todas las posibles soluciones. Sin embargo, como hemos ido mencionando previamente, el *backtracking* puede ser costoso en términos de tiempo y memoria, por lo que es importante elegir una estrategia de búsqueda y poda adecuada para optimizar el rendimiento del algoritmo.

Algunos casos en los que se puede utilizar el *backtracking*:

1. Problemas de búsqueda exhaustiva: El *backtracking* es útil para buscar todas las posibles soluciones a un problema, como encontrar todos los caminos posibles en un laberinto o buscar todas las posibles combinaciones de elementos de una lista. Un ejemplo a este tipo de problemas sería el famoso **problema de las N-Reinas**.

2. Problemas de optimización: El *backtracking* se puede utilizar para buscar la mejor solución a un problema, como encontrar la ruta más corta en un grafo o encontrar la combinación de elementos que maximice una función de utilidad. Un ejemplo a este tipo de problemas sería el famoso **problema de la mochila**.

3. Problemas de combinación y permutación: El *backtracking* es útil para encontrar todas las combinaciones y permutaciones posibles de un conjunto de elementos, como encontrar todas las permutaciones de una cadena de caracteres o todas las combinaciones posibles de un conjunto de números.

4. Problemas de asignación: El *backtracking* se puede utilizar para asignar recursos a tareas de manera óptima, como asignar trabajadores a tareas de manera que se maximice la productividad o asignar horarios a profesores de manera que se maximice la disponibilidad de tiempo.

V.V ANALISIS DEL COSTE

El *backtracking* puede presentar problemas de complejidad computacional. En general esta técnica produce algoritmos que presentan costes computacionales asintóticos **factoriales** o **exponenciales**. Por ejemplo, en algunos problemas de optimización, el número de posibles soluciones puede crecer exponencialmente con el tamaño del problema. Esto significa que el algoritmo puede tardar una cantidad de tiempo muy grande en encontrar la solución correcta.

V.VI NP-COMPLETO

Un detalle curioso es acerca de los programas NP-Completo.

Son aquellos que pertenecen a la clase de complejidad computacional NP y además es NP-Hard, lo que significa que no se conocen algoritmos polinómicos para resolverlos.

Al no poder conseguir esto, se encuentran dentro de los problemas que son impracticables que incluso en algunos casos no se pueden solventar.

Para solventar estos tipos de problemas existen varias técnicas para tratar de resolverlos o aproximarse a una solución.

Una de ellas sería la de *backtracking*, pero en esta debemos tener en cuenta que la complejidad es exponencial cosa que puede no ser práctico para problemas grandes. Para ello, en estos casos, es necesario utilizar técnicas más avanzadas como la programación dinámica o la heurística.

V.VII PROBLEMAS DE LAS N-REINAS

El problema de las N-Reinas [6] es un problema clásico de colocación de reinas en un tablero de ajedrez de tamaño NxN en el cual hay que encontrar una forma de colocar N reinas en de tal manera que ninguna reina esté en la misma fila, columna o diagonal que otra reina, es decir, que no se amenacen entre ellas.

Este problema se puede resolver utilizando el algoritmo de *backtracking*. Se trataría de un problema de búsqueda exhaustiva o decisión.

La idea es ir colocando las reinas en el tablero de forma recursiva, y si en algún momento se detecta que no es posible colocar una reina sin que amenace a otra, se realiza una vuelta atrás y se intenta con otra posición.

Este algoritmo es eficiente ya que no es necesario comprobar todas las posibles soluciones del espacio de búsqueda, sino que se van descartando aquellas que no son viables. Sin embargo, la complejidad del problema de las N-Reinas aumenta exponencialmente con el tamaño del tablero, lo que hace que no sea práctico para tamaños grandes.

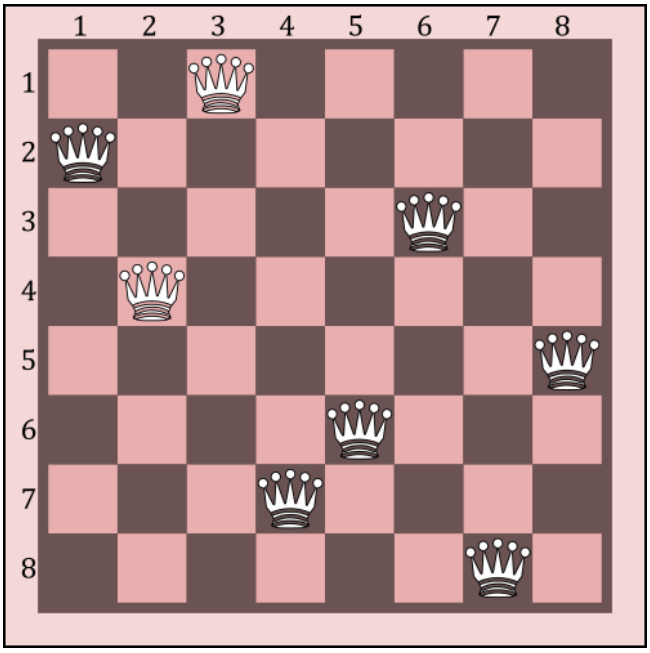

Figura 5: Problema de las N-Reinas

V.VIII PROBLEMAS DE LA MOCHILA

El problema de la mochila [7] es otro de los problemas clásicos de optimización combinatoria en el que se trata de encontrar la combinación de elementos que maximice el valor total sin superar una capacidad máxima determinada.

Disponemos de una colección de elementos y de una mochila. Cada elemento tiene asociados un peso y un valor. La mochila puede soportar un peso máximo. Se trata de encontrar la combinación de elementos colocados dentro de la mochila con lo cual consigamos el valor máximo, es decir, que hagamos que la mochila sea lo más valiosa posible cumpliendo la condición de peso máximo (no superar este peso).

Este problema se puede resolver utilizando el algoritmo de *backtracking*, que consiste en generar todas las posibles combinaciones de elementos que se pueden incluir en la mochila y elegir la mejor de ellas.

Este algoritmo de *backtracking* es eficiente para problemas pequeños, pero la complejidad del problema de la mochila aumenta exponencialmente con el número de elementos, por lo que no es práctico para problemas grandes.

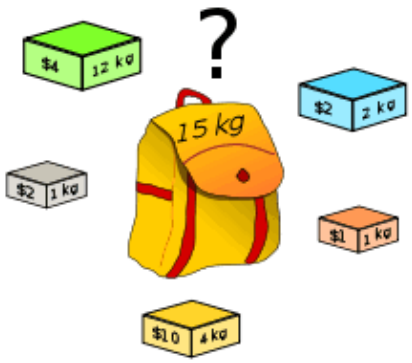

Figura 6: Problema de la mochila

VI. IMPLEMENTACIÓN MVC

Como ya hemos comentado anteriormente en este informe existen tres tipos de variaciones según respecta a la implementación del patrón "Modelo Vista Controlador". En esta práctica, debido a la facilidad que ofrece a la hora de gestionar los distintos módulos, así como para facilitar la compresión de los distintos componentes sin mezclar funcionalidades, se ha optado por desarrollar el programa siguiendo un esquema centralizado.

Para refrescar cabe comentar que este esquema persigue que la gestión de la comunicación entre las partes implicadas pase por un intermediario común. Este centro de control se responsabiliza también del ciclo de vida de los componentes.

El "mánager" central se ha implementado en la clase **Main,** a partir del cual empieza a ejecutarse el software una vez se inicia el programa.

Lo que permite que la vista, el controlador y el modelo se puedan comunicar entre sí es una interfaz llamada **EventListener** que obliga a implementar la función "**notify**". Mediante el uso de la clase abstracta **Event**, el **main** podrá distinguir cuál de los componentes ha notificado del evento y redireccionar el evento según corresponda. Esta solución es posible ya que cada componente implementa la misma interfaz, por lo que también deben implementar su propia versión de "**notify**", y esta será llamada desde el **main** cuando éste decida que un evento debe ser tratado por la correspondiente parte. Por tanto, en el **main** tendrá una correspondencia a cada componente de la arquitectura.

La arquitectura descrita permite una gran versatilidad a la hora de implementar los distintos elementos por separado debido a que existe un gran desacople en el sistema de comunicación. Esto permite desarrollar un módulo sin tener que pensar en cómo afectará eso en la gestión global.

VI.I MODEL

El modelo es una parte fundamental de cualquier aplicación, ya que es el componente encargado de almacenar y gestionar todos los datos que serán escritos o leídos por los usuarios, tanto de manera directa como indirecta. En otras palabras, el modelo es el lugar donde se guardan y organizan los datos que serán utilizados para realizar diferentes operaciones y análisis en la aplicación.

En el caso específico de nuestra aplicación, el modelo estaría compuesto por una matriz de objetos celda para representar el tablero, estructuras n-en las que almacenar las piezas presentes en dicho tablero y un contador de movimientos totales realizados sobre el tablero.

BoardCell

Cada celda viene representada por una instancia de la clase BoardCell, la cual almacena un índice a la posible pieza que ha aparecido en la celda y el número de movimiento con el que la pieza ha pasado por la celda.

Por *default*, a una celda vacía se le asigna -1 para ambos atributos, indicando así que no ha pasado aún ninguna pieza.

Piece

La clase Piece es una clase *abstract*, es decir, no se pueden crear instancias de ella directamente, sino que en su lugar se deben crear clases que extiendan esta clase y proporcionen la funcionalidad específica para cada diferente tipo de pieza de ajedrez.

Esta clase contiene varios campos de datos protegidos (*protected*) que puedes ser accesibles únicamente por las clases "hija" o subclases, y varios métodos para acceder y establecer lo diferentes atributos (*getters* and *setters*).

Estos campos protegidos son:

a) **movx[]**: es un array de enteros que representa los movimientos en el eje X que puede hacer la pieza.
b) **movy[]**: lo mismo pero respecto al eje Y.
c) **name**: es una cadena de caracteres que representa el nombre de la pieza.
d) **image**: es una cadena de caracteres que representa el nombre de la imagen de la pieza.
e) **affectsdimension**: un booleano que indica si la dimensión del tablero afecta a la movilidad de la pieza.
f) **posx y posy:** enteros que representan la posición actual de la pieza en el tablero.

Los métodos son:

a) **afectaDimension():** devuelve el booleano **affectsdimension**.
b) **getName()**: devuelve el nombre de la pieza.
c) **getImage()**: devuelve el nombre de la imagen de la pieza.
d) **getNumMovs()**: devuelve la cantidad de movimientos que puede realizar la pieza.
e) **getMovX(int i) y getMovY(int i)**: devuelve el movimiento en el eje X e Y de las respectivas array en la posición i.
f) **setPos()**: establece la posición actual de la pieza en el tablero.
g) **getPosX() y getPosY()**: devuelve la posición actual de la pieza en el eje X y eje Y respectivamente.
h) **getPiecesTypes()**: un método estático que devuelve un arreglo de cadenas de caracteres que representa los tipos de piezas disponibles. Los valores del arreglo se obtienen a partir de un enum llamado "PieceTypes".
i) **PieceTypes**: enum que contiene los nombres de los tipos de piezas disponibles.

Estos dos últimos componentes son de utilidad para uno de los objetos que conforman la vista, una lista que muestra los tipos de pieza que existen y que aparecerá explicado más adelante.

Las diferentes piezas que existen [8] son:

- Horse → se mueve dos casillas en horizontal o vertical y después una casilla más en ángulo recto. Simula un movimiento en forma de “L”.
- Queen → se mueve tanto en horizontal como en vertical como en diagonal a lo largo de la dimensión n del tablero.
- Tower → se mueve en una línea horizontal o vertical a lo largo de la dimensión n del tablero.
- King → se mueve a cualquier casilla adyacente.
- BicMac → se trata de una versión “mejorada” del rey que en vez de poder solo irse a las casillas adyacentes, una posición, puede hacer avanzar dos posiciones.
- Bishop → se mueve en una línea diagonal a lo largo de la dimensión n del tablero.

Para aquellas piezas con el atributo **affectsdimension** a True, hay un segundo constructor que acepta como parámetro un entero para establecer la dimensión del tablero, la cuál afecta a la cantidad de posibles movimientos que podrá realizar cada pieza.

ModelEvent

La clase ModelEvent hereda de **Event** y se utiliza para que el **main** notifique al Model cuando se produzca un evento que este deba gestionar. Además, aparece un enumerado ModelEventType para diferenciar el tipo de ModelEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro del Modelo.

El ModelEventType compone los eventos de modelo:
- SET_DIMENSION
- START
- MOVE_PICE
- ADD_SELECTED_PIECE
- PRUNE
- RESET

Los atributos de ModelEvent son todos aquellos necesarios para realizar la correcta gestión y actualización respecto a la llegada de un evento determinado.

Model

Junto a las ya mencionadas matriz de BoardCell y contador de movimientos totales, se encuentran dos estructuras para almacenar objetos de tipo Piece: un ArrayList y un array. El equipo ha tomado la decisión de utilizar un ArrayList en la que ir añadiendo las piezas a medida que el usuario vaya añadiendo instancias en el tablero, para así una vez se tenga que iniciar la ejecución del algoritmo, previamente hagamos un volcado de las instancias del ArrayList al array para que cuando el algoritmo tenga que acceder a los objetos Pieza almacenados en la estructura, pueda hacerlo directamente, dada la naturaleza de la estructura array.

El constructor de la clase recibe el **main,** referencia al programa principal para realizar las comunicaciones entre componentes; y se inicializan las estructuras de la clase en función del entero pasado por parámetro y que representa la dimensión que debe tener el tablero. El tablero se construye a partir de instancias BoardCell vacías.

Se han implementado una serie de métodos para configurar y actualizar el estado del modelo y unos *getters* necesarios tanto para el propio modelo como para el controlador para obtener información sobre el estado del modelo. Los dos que son más interesantes para destacar son getPieceTurn() y getMovement(). Ambos usan la variable de conteo de movimientos y el número de piezas existentes en el tablero. El primero, para obtener el turno realiza la operación módulo de la cantidad de movimientos por la cantidad de piezas. El segundo, para obtener el movimiento a calcular, realiza una división entera entre los dos mismos operadores.

Otros métodos que son interesantes son:
- isValidMovement(int x, int y): comprueba si la casilla de la posición indicada por parámetro se encuentra dentro del tablero y vacía.

- isFreeCell(int x, int y, int movement, int pieceIndex): no solo tiene en cuenta si la celda puede estar libre, también considera como válidas aquellas celdas en las que hay alguna pieza con menor índice de prioridad y con un movimiento superior al pasado por parámetro.
- prune(int movementToPrune, int pieceIndex): reinicia las casillas con movimientos superiores al pasado por parámetro, diferenciando los casos de piezas con menor y con mayor índice de prioridad en comparación al pasado por parámetro.
- addPiecePlayer(String name, int x, int y): genera una instancia de una de las clases de Pieza en función de la String name pasada por parámetro. Si la clase tiene más de un constructor, quiere decir que es una de esas piezas que se ve afectada por la dimensión del tablero; por lo tanto, al constructor se le pasaría por parámetro dicha dimensión.

La clase Model implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realizan unas operaciones u otras en función del ModelEventType recibido. START hace el volcado de instancias del ArrayList de Pieza a un array, SET_DIMENSION reconstruye el tablero con la nueva dimensión almacenada por el evento, MOVE_PIECE sitúa una pieza en una casilla, ADD_SELECTED_PIECE instancia un nuevo objeto Pieza de la clase requerida y se la sitúa en la casilla indicada; PRUNE volvería atrás las posiciones de las piezas en el tablero hasta un estado previo del que seguir generando posibles estados descendientes válidos; y finalmente RESET realiza una limpieza de las piezas de las estructuras correspondiente y se reconstruye el tablero con casillas vacías.

VII.II CONTROLLER

El controlador es el componente con el rol de bisagra situada entre la vista y el modelo de datos previamente explicado. Es decir, es el encargado de gestionar el flujo de datos entre los dos módulos comentados. Este flujo de datos es transformado mediante operaciones ejecutadas por el mismo controlador y visualizado en la vista o almacenado en el modelo.

En nuestra aplicación el controlador se responsabiliza de ejecutar el *backtracking* correspondiente para recorrer todas las celdas de forma **única**.

PieceState

Esta clase se utiliza para representar el estado de una pieza durante la ejecución del algoritmo. Se usan instancias de esta clase para ser almacenadas en las distintas pilas y conseguir así simular el efecto de la recursividad cuando se ejecute el algoritmo de *backtracking*.

Un objeto PieceState guarda información de la posición a la que se ha de desplazar en un cierto movimiento la pieza en cuya pila se encuentra dicho estado.

De tal manera que el constructor tiene la forma:
PieceState(int x, int y, int movement)

ControllerEvent

La clase ControllerEvent hereda de **Event** y se utiliza para que el **main** notifique al Controller cuando se produzca un evento que este deba gestionar. Además, aparece un enumerado ControllerEventType para diferenciar el tipo de ControllerEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro del Controller.

El ControllerEventType compone los eventos de modelo:
- SET_SPEED
- START

ControllerEvent únicamente necesita como atributos uno que haga referencia a la *speed* de ejecución del algoritmo y otro para almacenar el tipo de evento de control.

Controller

En primer lugar, se obtiene una instancia del modelo, que es la clase que representa el estado actual del problema. A continuación, se inicializa una lista de pilas de estados para cada pieza del problema y mientras no se encuentre la solución o no se haya determinado que no hay solución se realizan las siguientes operaciones:

- Obtención del turno de la pieza actual.
- Obtención de la pila de estados actual para la pieza actual.
- Generación de los posibles movimientos de la pieza actual y almacenamiento en la pila de estados si son válidos.
- Si no hay movimientos válidos, determinación de que no hay solución.
- Si hay movimientos válidos, obtención de un estado válido de la pila.
- Podar estados no necesarios de la pila actual y del resto de pilas.
- Si el movimiento actual es menor al movimiento máximo permitido, se notifica a la vista que se deben podar celdas.
- Notificación a la vista de la nueva posición de la pieza y repintado.
- Si todas las celdas del tablero están ocupadas, se ha encontrado una solución.
- Se espera un tiempo para mejorar la visualización del algoritmo.

Finalmente, por motivos de depuración de código se muestra el número de celdas visitadas y a continuación se notifica a la vista el resultado final del algoritmo.

En general, se implementa un algoritmo de búsqueda en profundidad con poda de estados no necesarios.

La clase Controller implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realizan unas operaciones u otras en función del ControllerEventType recibido. START simplemente ejecuta el Thread del controlador para que empiece el algoritmo y SET_SPEED asigna al atributo estático de la clase Controller el nuevo valor entero que se sido recibido por el evento.

VII.III VIEW

La vista representa la interfaz de usuario de la aplicación y se encarga de presentar los datos al usuario final. La vista no interactúa directamente con el modelo ni con el controlador, sino que se comunica con ellos a través del gestor de eventos.

BoardDisplay

El constructor de esta clase recibe como parámetro una referencia al Model para facilitar la consulta del modelo y poderlo pintar de manera adecuada. En el constructor también se inicializa un HashMap<Integer, Color> para asignar a cada pieza representada por un índice un color aleatorio. Como consecuencia, cuando se pinte el tablero se podrá distinguir los movimientos que realiza una pieza u otra.

El método paint() de la clase recibe la dimensión del tablero para dibujar una matriz de cuadrados, simulando un tablero. Mediante el modelo, comprobaremos si una casilla se encuentra ocupada por una pieza o no. En caso afirmativo, sobre la celda se pinta la imagen y el movimiento correspondiente a la pieza y que devuelve el modelo en forma de String y de entero respectivamente. Consultando la estructura hashMap y habiendo obtenido el índice que representa a la pieza, se pintará el movimiento de la pieza con un nuevo color que la pueda distinguir del resto de colores de piezas. Cabe decir que en ningún momento se asegura que estos colores sean más o menos distintos, si no que simplemente se calcula cada componente del color usando un *random* de 0 a 255.

ViewEvent

La clase ViewEvent hereda de **Event** y se utiliza para que el **main** notifique a la View cuando se produzca un evento que esta deba gestionar. Además, aparece un enumerado ViewEventType para diferenciar el tipo de ViewEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro de la View.

El ViewEventType compone los eventos de modelo:

- REFRESH_BOARD
- ALGORITHM_END

ViewEvent únicamente necesita como atributos uno que haga referencia a si la terminación del algoritmo ha sido satisfactoria o no y otro para almacenar el tipo de evento de vista.

View

La View envía eventos tanto al Controller como al Model. Notifica al Controller cuando se modifique la velocidad de ejecución del algoritmo y cuando se pulse el botón para que se inicie la ejecución de este. Al Model se le notifica cuando se coloque una nueva pieza sobre el tablero, cuando se modifique el tamaño del tablero y para que se vuelquen las piezas de una estructura a otra previamente al inicio de la ejecución del algoritmo.

La vista es responsable de presentar los datos de manera clara y coherente para el usuario. Esto incluye la disposición de la información, la selección de la paleta de colores, la presentación de los datos en diferentes formatos y la gestión de la interacción del usuario, además de poder apreciar el correcto comportamiento del algoritmo con el cual se visualiza el movimiento de las piezas por el tablero.

Podemos apreciar todos estos elementos en la interfaz:

1. JSpinner: se inicializa con el valor de dimensión por *default* y al modificar su valor se notifica al modelo cuál debe ser la nueva dimensión del tablero para posteriormente refrescar la vista del tablero. No acepta valores menores de 2.

2. JScrollPane: recibe como lista de elementos los nombres de las clases de Pieza existentes, es decir se utiliza del método estático de Pieza que retorna los nombres de la clase enumerada de tipos de pieza. Si se quisiera añadir una nueva clase de Pieza, bastaría añadir el nombre en el enumerado para que se viese reflejado en la vista. Este componente es útil para que el usuario pueda elegir que tipo de pieza de entre los existentes quiere disponer en el tablero.

3. JSlider: tiene definidos unos valores máximos y mínimos para elegir la velocidad de ejecución del algoritmo. Al modificar su valor, se notifica al controlador y se muestra en el campo de texto situado encima suyo.

4. JTextField: también en este campo de texto se puede

establecer una nueva velocidad para el algoritmo, respetando los mismos límites definidos por el *slider*. Al mismo tiempo, notifica la nueva velocidad al Controller.

5. JButton (Start): solo permite pulsar el botón mientras haya alguna pieza en el tablero, siempre y cuando no haya ninguna ejecución del algoritmo iniciada. Cuando el botón sea pulsado, enviará los correspondientes eventos al modelo y al controlador, asegurando la consistencia del resto de elementos de la vista.

6. JButton (Reset): este botón está solo disponible para ser pulsado cuando haya finalizado completamente la ejecución del algoritmo. Notifica al Model que debe hacer el *reset* del tablero y posteriormente manda el *refresh* del *display* del tablero. Igual que el botón de *start*, se asegura de la consistencia del resto de elementos de la vista.

7. BoardDisplay: el usuario podrá colocar en el tablero la pieza seleccionada de la lista. En función de las coordenadas en las que haya hecho el clic, se podrá notificar al Model en qué posición deberá situar la nueva pieza. Posteriormente, se hará un *refresh* del *display* del tablero.
   Para evitar colocar más de una pieza sobre la misma casilla, se consultará si la casilla en cuestión ya ha sido ocupada o no.
   También se puede mencionar que cuando el usuario pueda colocar una pieza sobre el tablero, el cursor del ratón aparecerá con la 'forma de mano'.

Los elementos de la vista previamente descritos se pueden observar en la Figura 7.

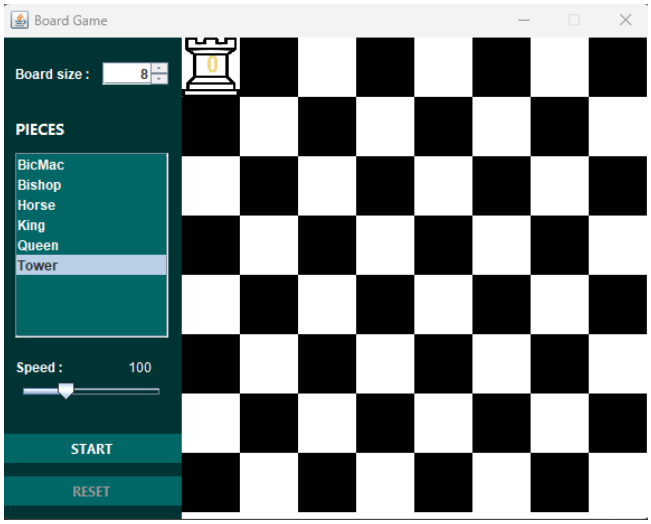

Figura 7: Vista de la aplicación

La clase View implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realizan unas operaciones u otras en función del ViewEventType recibido. REFRESH_BOARD simplemente manda el *refresh* del *display* del tablero y ALGORITHM_END para habilitar el botón de *reset* y para mostrar por pantalla un mensaje con la resolución del algoritmo, indicado si una solución ha sido encontrada o no.

VIII. CONCLUSIÓN

El equipo considera que la implementación del algoritmo de *backtracking* de la práctica utilizando el patrón de diseño MVC ha sido exitosa en términos de estructura y eficiencia. Pensamos que el uso del MVC junto a su consecuente abstracción ha sido bastante mejor en comparación a la realizada en la primera práctica. Esto nos ha permitido separar la lógica del algoritmo de la interfaz de usuario sin interferir unas con otras, lo que ha resultado en un código más organizado y fácil de mantener. Además, la utilización de *backtracking* ha permitido encontrar soluciones para un problema que requería una exploración exhaustiva de todas las posibilidades. En general, la combinación de ambos elementos ha resultado en un programa robusto y escalable.

En un primer momento pensamos en usar un algoritmo de *backtracking* recursivo, pero la idea del uso de valores muy elevados para el tamaño del tablero nos hizo desechar la idea ya que supondría un gran gasto de memoria. A continuación, desarrollamos la idea que al final ha sido definitiva, el uso de un *backtracking* iterativo con la implementación de una pila para cada pieza. Gracias a este método obtenemos resultados a gran velocidad, pero con la desventaja del gasto de memoria. Finalmente nos dimos cuenta de que era posible implementar el mismo algoritmo con el uso de una única pila global, la solución óptima del problema, pero con el inconveniente de los pocos días que quedaban para la entrega no pudimos finalizar su desarrollo. Cabe destacar que la solución proporcionada es bastante similar a la óptima y que nuestro razonamiento, pese a tener el inconveniente de la memoria, sigue una lógica válida.

Siendo nuestro segundo trabajo juntos, en el que hemos podido reforzar la implementación del MVC, el equipo ha funcionado bastante bien. El control de versiones mediante GitHub ha sido mucho más utilizado y ha ganado mucha más utilidad en esta práctica que en la anterior, en la cual esto nos supuso algunos problemas. Por los problemas que respectan a esta práctica, ha sido la gran variedad de opciones y estilos de implementaciones que hemos intentado llevar a cabo, cosa que pensamos que al final nos ha acabado pasando factor. A pesar de ello, creemos que el resultado final conseguido por el equipo ha sido bastante satisfactorio.

IX. ENLACES

1. Teoría del patrón MVC: https://developer.mozilla.org/en-US/docs/Glossary/MVC

2. Teoría del patrón por eventos: https://www.tibco.com/es/reference-center/what-is-event-driven-architecture

3. Ventajas y desventajas del patrón por eventos: https://es.wikipedia.org/wiki/Arquitectura_dirigida_por_eventos

Backtracking:

4. https://www.apascualco.com/algoritmos/algoritmo-de-backtracking/
5. https://docs.jjpeleato.com/algoritmia/backtracking

Problema de la mochila:

6. https://www.discoduroderoer.es/el-problema-de-la-mochila-en-java-con-backtracking/

Problema de las N-Reinas:

7. https://developers.google.com/optimization/cp/queens

Recursividad:

8. https://www.campusmvp.es/recursos/post/Que-es-la-recursividad-o-recursion-Un-ejemplo-con-JavaScript.aspx

Piezas ajedrez:

9. https://docs.kde.org/trunk5/es/knights/knights/piece-movement.html#:~:text=El%20caballo%20se%20mueve%20dos,la%20de%20su%20casilla%20inicial.