Capitulo 2

Jonathan Zinzan Salisbury Vega
Universitat de les Illes Balears
Dpto. Ingeniería Informática
Palma, España
jonathan.salisbury1@estudiant.uib.cat

Joan Sansó Pericás
Universitat de les Illes Balears
Dpto. Ingeniería Informática
Palma, España
joan.sanso4@estudiant.uib.cat

Joan Vilella Candia
Universitat de les Illes Balears
Dpto. Ingeniería Informática
Palma, España
joan.vilella1@estudiant.uib.cat

Los problemas de satisfacción de restricciones han sido siempre unos de los más fáciles de comprobar, pero los más difíciles de resolver. Se hablará de una de las técnicas más usadas para este tipo de problemas, el backtracking, y sus características. Se hablará del problema del recorrido del caballo en un tablero de ajedrez y de sus variantes con otras piezas, así como de nuestra solución con backtracking, un estudio de las complejidades asintóticas y del rendimiento.

Keywords—Backtracking, recursividad, ajedrez, coste asintótico

I. INTRODUCCIÓN

En este documento se pretende explicar los conceptos relacionados con el tema dos de la asignatura de algoritmos avanzados. El principal tema de estudio es el backtracking, aunque se explicarán los distintos costes asintóticos de los algoritmos implementados.

Por otra parte, está la solución implementada para el problema del recorrido de una pieza sobre un tablero NxN. Para ello se han tenido que implementar una serie de piezas inventadas para cumplir con los requisitos de la práctica. Con motivo de este problema también se ha aprovechado para indagar en lo teoría relacionada con los recorridos hamiltonianos que es, al fin y al cabo, lo que se está implementando en esta práctica.

También se ha aprovechado para dar contexto al Backtracking, explicando antes que es un tipo de recursividad. O, mejor dicho, que se suele utilizar la recursividad a la hora de aplicar este tipo de algoritmos, ya que quedan estructurados de una manera muy lógica de cara al programador. Lo que no suele ser siempre la mejor opción en cuanto a eficiencia, por eso se suelen pasar a iterativos este tipo de algoritmos.

Finalmente, toda esta práctica se ha de desarrollar siguiendo el patrón de desarrollo de software MVC (Modelo Vista Controlador). Para ello se ha tenido que desarrollar la práctica teniendo en cuenta estas tres clases que determinan este comportamiento.

II. CONTEXTO Y ENTORNO DE ESTUDIO

Uno de los requisitos a la hora de realizar la práctica es el uso del lenguaje de programación Java. Además, se ha dado la opción de elegir entre dos IDE (Integrated Development Environment).

- NetBeans
- IntelliJ

En este caso se ha escogido el IDE de NetBeans por familiaridad de uso. Además, se utiliza una herramienta de control de versiones (Git). Más específicamente su versión de escritorio Github Desktop por su facilidad de uso mediante interfaz gráfica.

III. DESCRIPCION DEL PROBLEMA PROPUESTO

El problema que se plantea en esta práctica es una variante del recorrido de una pieza de ajedrez en un tablero de tamaño *NxN*. El problema original consiste en encontrar un camino en el que la pieza (usualmente un caballo) visite exactamente una vez cada casilla del tablero.
Este es un problema matemático utilizado comúnmente para aprender y aplicar conceptos de programación y desarrollo de algoritmos. En este caso se especifica que debe poder resolverse con seis piezas cada una con movimientos distintos y desde una posición inicial en el tablero arbitraria. El algoritmo también debe notificar en caso de que no exista una posible solución al problema dado.

Como es fácil de observar, la descripción del problema encaja a la perfección con la definición del recorrido Hamiltoniano de la teoría de grafos. En este caso cada casilla del tablero será un vértice del grafo a recorrer y las aristas que conecten a estos dependerán de los movimientos de la pieza seleccionada.

A. Camino Hamiltoniano

Es importante dar una primera definición formal de lo que se entiende por camino hamiltoniano:
"*En el campo matemático de la teoría de grafos, un camino*[1] *hamiltoniano es un camino en un grafo (dirigido o no) en el que se visita cada vértice exactamente una vez.*"

[1] Sucesión de vértices y aristas dentro de un grafo (ha de comenzar y terminar en un vértice).

Dentro de los caminos hamiltonianos existe un subconjunto que son los ciclos hamiltonianos. Estos son más restrictivos ya que han de comenzar y acabar en el mismo vértice.

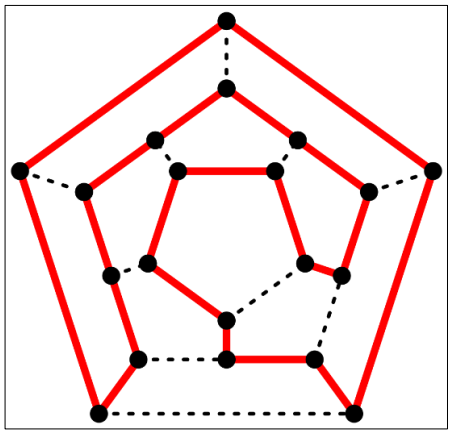

Ilustración 1: El grafo del dodecaedro es hamiltoniano, como el resto de los sólidos platónicos.

Encontrar un ciclo hamiltoniano en un grafo cualquiera puede parecer una tarea a priori sencilla, pero esto está muy alejado de la realidad. Dicho problema se sabe que es NP-Completo[2] y además está listado como uno de los 21 problemas NP-Completos de Karp[3].
A continuación, se revisarán las condiciones necesarias y suficientes para que pueda existir un camino hamiltoniano.

1. *Condiciones necesarias*
   - Un grafo hamiltoniano ha de ser conexo.
   - Un grafo hamiltoniano no puede tener vértices de grado 1.
   - Un grafo hamiltoniano no puede poseer vértices de corte, es decir, un vértice tal que, si se elimina, el grafo resultante tiene más componentes conexas.

2. *Condiciones suficientes*
   Por motivos de complejidad matemática y que dichos teoremas quedan fuera del alcance de la asignatura, sólo se mencionaran dichos teoremas que son las condiciones suficientes:

   - Teorema de *Bondy-Chvátal.*
   - Teorema de *Ore.*
   - Teorema de *Dirac.*

B. *Piezas Disponibles*

El problema que se ha descrito antes deberá ser resuelto con una serie de piezas típicas del ajedrez, además de unas nuevas propias inventadas para el problema. A continuación, se describirán dichas piezas:

1. *Caballo:* Pieza original del problema descrito. Su movimiento consta de un desplazamiento de dos casillas de manera horizontal y una vertical o dos casillas de manera vertical y una horizontal. Comúnmente se describe con un movimiento similar a una "*L*".
2. *Torre:* La torre se mueve de manera vertical u horizontal a lo largo de cualquier número de casillas desocupadas.
3. *Rey:* El rey se puede mover a cualquier casilla adyacente pero solo una casilla.
4. *Reina:* Es la pieza con mayor movilidad del juego, se puede mover tanto de manera horizontal como vertical a lo largo de todas las casillas en línea recta.
5. *Conejo:* Esta pieza inventada tiene similitud con el caballo porque también salta casillas. En este caso el conejo siempre se mueve de dos en dos casillas.

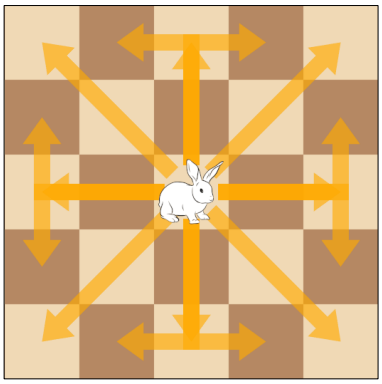

Ilustración 2 Movimientos del Conejo en el Tablero

6. *Elefante:* Esta pieza se podría decir que es un rey vitaminado. Tiene el mismo rango de movimientos a excepción de cuando se mueve de manera vertical u horizontal que, en ese caso, se puede mover dos posiciones.

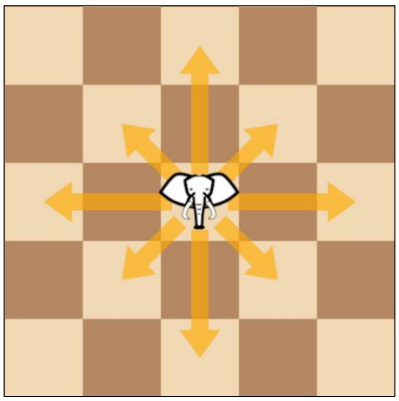

Ilustración 3 Movimientos del Elefante en el Tablero

7. *Bota:* Esta pieza tiene el mismo concepto que el conejo, pero con una casilla más de rango, es decir, se desplaza en saltos de tres casillas.

[2] Es un problema para el cual verificar solución es muy fácil, pero para encontrarla se necesitan algoritmos de fuerza bruta.

[3] Lista elaborada por el informático Richard Karp, en la que recoge una serie de problemas famosos en la que todos son de complejidad NP-completos.

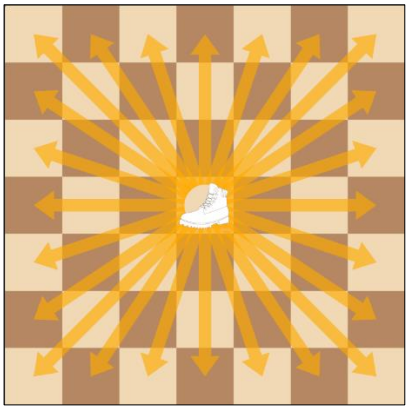

Ilustración 4 Movimientos de la Bota en el Tablero

A la hora de diseñar las nuevas piezas uno de los criterios fundamentales que se han tenido en cuenta es que el movimiento ha de permitir cambiar de color de casilla. Una vez entendido es trivial (porque si se piensa en el alfil es imposible que pueda recorrer todas las casillas).

IV. SOLUCIÓN MEDIANTE RECURSIVIDAD

La recursividad es la forma en la cual un proceso se especifica basado en su misma definición. Más concretamente, en programación, es un método usual de resolver un problema de forma sencilla, como puede ser el problema de las Torres de Hanoi[4], recorrido de Grafos, la búsqueda Euleriana…
Otros ejemplos de algoritmos que se definen de forma recursiva por su naturaleza son: los Números de Fibonacci, la función factorial. A continuación, se van a explicar los distintos tipos de recursividad.

A. Tipos de recursividad

Dependiendo de la estructura de una función recursiva se puede clasificar mediante las siguientes categorías:

1. *Recursividad Directa*
   La recursividad directa se define como una función quese llama a sí misma, con código a ejecutar tanto antes como después de la llamada recursiva.
2. *Recursividad Indirecta*
   Se llama recursividad indirecta cuando una función no se llama a sí misma, sino que llama a otra función que puede llegar a desencadenar una llamada recursiva (es decir: función 1 llama a función 2, función 2 llama a función 3, función 3 llama a función 1: hay recursión, pero de forma indirecta).
3. *Recursividad de cola*
   Es un tipo de recursividad directa donde la llamada recursiva es la última sentencia de la función (las operaciones y cálculos se realizan antes de la recursión).
4. *Recursividad de cabeza*
   Como la recursividad de cola, es un tipo de recursividad directa, pero con la diferencia que la llamada recursiva es la primera sentencia de la función. Las operaciones se realizan cuando la recursión ha llegado al caso base y las llamadas recursivas empiezan a volver.
5. *Recursividad Lineal*
   Es un tipo de recursión donde la cantidad de llamadas es proporcional al tamaño del problema, es decir, crece de forma lineal con la entrada (una sola llamada recursiva por función, p. ej.: Factorial).
6. *Recursividad en Árbol o no Lineal*
   A diferencia de la Lineal, en estas funciones se hace más de una llamada recursiva en la función y, por ende, la recursión crece en forma de árbol (p. ej.: Fibonacci, donde se hacen 2 llamadas recursivas en cada llamada, o en nuestro algoritmo de *Backtracking*, donde se pueden llegar a realizar tantas llamadas como movimientos posibles tiene una pieza).

El concepto de recursividad tiene algunas ventajas y desventajas según el caso en el que se quiera utilizar, a continuación, se describen las principales:

- *Ventajas:*
  La recursividad puede **reducir la complejidad temporal**. Esto puede parecer contraintuitivo, pero si aplicas programación dinámica y **memorizas** soluciones parciales, puedes disminuir la complejidad temporal. Por ejemplo, para el cálculo de los números de Fibonacci, donde el algoritmo que memoriza es simplemente de complejidad $O(n)$ tanto temporal como espacialmente (si se usa la estructura de datos adecuada).

  Los programas recursivos suelen ser más “simples” y suelen tener menos líneas de código.
  Son muy buenos para recorrer árboles. Debido a esto, se usan para implementar recorridos de árboles como el Preorden y para la generación de éstos (*véase Backtracking*).

- *Inconvenientes*
  Si la cantidad de llamadas seguidas es muy grande puede dar problemas de memoria, ya que las llamadas van llenando la pila del sistema, y puede haber un error de *Overflow*[5], que será aún más rápido si se usan muchas variables locales, ya que en cada llamada recursiva éstas se tienen que guardar, ocupando aún más espacio.

  Si se implementa mal, o se usa la recursividad en problemas sencillos, puede ser muy lenta. Esto es debido a que la recursividad necesita más tiempo al hacer llamadas y volver de estas debido a todo el manejo de memoria que se tiene que hacer, mientras que las iterativas son simples bucles sobre los mismos datos.

[4] Juego de mesa individual consiste en un número de discos perforados de radio creciente que se apilan insertándose en uno de los tres postes fijados a un tablero.

[5] Es un error que ocurre cuando un programa recibe un número, valor o variable fuera del alcance de su capacidad de manejo.

Los conceptos anteriores se pueden aplicar a nuestro problema mediante la aplicación de un tipo de algoritmo recursivo llamado *Backtracking*.

B. Backtracking

El *backtracking* es un algoritmo general para encontrar soluciones a problemas que tienen que satisfacer un conjunto de condiciones. La traducción literal del nombre seria "Paso atrás", y se le da ese nombre debido a la naturaleza de estos algoritmos: van creando candidatos a la solución de manera incremental, y abandonan un candidato (dan un paso atrás) cuando se determina que ese candidato no podría ser solución.
De esta manera, se va generando un *árbol de soluciones parciales*, y el algoritmo **poda** la rama del árbol cuando ve que la solución parcial no podrá llegar a una solución completa.

El algoritmo puede ser programado para que pare al encontrar una solución, o puede ser creado de forma que encuentre todas las soluciones, que pare cuando haya pasado un tiempo determinado, o haya gastado una cantidad de recursos (ciclos de CPU, memoria) determinados.

Al ser un subconjunto de los algoritmos recursivos, se aplican las ventajas y desventajas de éstos. Adicionalmente, podemos enumerar:

- *Ventajas*
  El backtracking, al ser un algoritmo que recorre todo el árbol de posibilidades, siempre va a ser capaz de encontrar todas solución, si es que existe alguna. De la misma manera, es capaz de asegurarte que no existe ninguna solución al conjunto de restricciones o condiciones establecidos.
  Para según que tipos de problemas, es mucho más sencillo programar un algoritmo recursivo que intentar hacer uno iterativo para el mismo objetivo. Esto es debido a la facilidad de aplicar el concepto de recursión en problemas que necesitan satisfacer una lista arbitraria de condiciones, ya que puedes hacer que cada llamada recursiva intente encontrar una solución parcial a una de las condiciones, y si llegas a un estado que no podrá desarrollarse en solución final, vuelves atrás e intentas de nuevo.

- *Inconvenientes*
  El coste asintótico de estas funciones suele ser del orden factorial ($O(n!)$) o exponencial ($O(a^n)$). Esto hace que para según qué problemas de gran tamaño, sea impracticable en el tiempo.

C. Aplicar Backtracking

Ahora que se entiende que es la recursividad y el *backtracking*, se procederá a explicar cómo se ha aplicado para solucionar el problema.

El algoritmo que se ha desarrollado hace lo siguiente: dada una posición (***x, y***), prueba uno a uno los movimientos de la pieza seleccionada desde esa posición, verificando si el movimiento es legal (es decir, si el movimiento cae dentro del tablero, y si es en una casilla que aún no se ha visitado). Si es así, pone el movimiento en el tablero y se llama recursivamente con la nueva posición.

Si en una llamada recursiva el número de movimiento es igual al número máximo de movimientos (es decir, la cantidad de casillas) significa que habremos llegado a nuestro caso base (todas las casillas han sido visitadas) y devolveremos *true*, lo que desencadenará la vuelta de todas las llamadas recursivas y significará que hemos encontrado una solución.

Si una llamada recursiva devuelve *false* significará que ésta ha probado todos los movimientos en su nivel y subniveles y ninguno ha dado una solución, así que intentaremos el siguiente movimiento hasta quedarnos sin movimientos, momento en que devolveremos *false* y se repetirá este proceso en una llamada superior.
Si la primera llamada ha recorrido todos los posibles movimientos y ninguno ha dado solución, significa que ésta no existe.

D. Complejidad de la Solucion

El coste asintótico de este problema es del siguiente orden:

$$O(m^{N^2})$$

Ecuación 1

Donde *N* es el tamaño del tablero (p. ej.: 8) y *m* la cantidad de movimientos posibles en cada llamada recursiva. Este número cambiará dependiendo de la pieza (p. ej.: un caballo tiene 8 posibles movimientos, así que la *m* será 8. Un rey también tiene 8. Una reina podría llegar a tener 56 movimientos posibles (aunque no todos son realizables en todos los casos).

Este orden de complejidad se saca de forma muy sencilla: si *m* es el número máximo de movimientos que puede hacer una pieza, y tenemos N·N casillas (N^2), el número de movimientos que se harán, en peor caso serán *m* movimientos en cada casilla, es decir:

$$\underbrace{m \cdot m \cdot m \cdot m \ldots}_{N^2 \text{ veces}} = m^{N^2}$$

Ecuación 2

Como se puede observar, este orden de complejidad crece peor que los exponenciales, ya que la N está al cuadrado en el exponente. Debido a esto, aunque la cantidad de movimientos posibles de la pieza sea muy pequeña, por ejemplo, 2, la alta complejidad hace que sea impracticable ($2^{8\wedge 2} \approx 18 \cdot 10^{19}$)

V. PATRON MVC

El Modelo-Vista-Controlador (MVC) [1] es un patrón de diseño de software[6] en el que se divide la lógica del programa de su representación gráfica, además se hace uso de un controlador para los eventos y comunicaciones entre las distintas partes.

Este patrón de arquitectura se basa en las ideas de reutilización de código y la separación de conceptos distintos, estas características pretenden facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento. Para ello se proponen la construcción de tres componentes:

- *Modelo*
  Es donde se almacena la información necesaria para la ejecución de la aplicación. Gestiona el acceso a dicha información mediante peticiones a través del controlador. También contiene los procedimientos lógicos que hagan uso de esa información.
- *Vista*
  Presenta la información en un formato adecuado para la interacción del usuario con los datos.
- *Controlador*
  Responde a eventos usualmente generados por el usuario y los convierte en comandos para la vista o modelo. Se podría decir que hace la función de intermediario entre la vista y el modelo encargándose de la lógica del programa.

Como se ha mencionado anteriormente se ha decidido utilizar este patrón debido a la facilidad que aporta al programar la separación de conceptos. Además, se obtiene una capa más de abstracción ya que es posible utilizar diferentes vistas con un mismo modelo.

Hoy en día, es muy común el uso de la programación orientada a objetos (POO) como paradigma principal, por ello cada componente del patrón MVC suele implementarse como una clase independiente. A continuación, se explica que proceso se ha seguido y como se ha implementado cada parte.

A. Implementacion del Modelo

Se ha diseñado la clase *Model* que se encarga de almacenar todos los datos relacionados con el tablero y la pieza de ajedrez seleccionada para el problema. Además, contiene los métodos necesarios para implementar el algoritmo que obtiene la solución al problema y los que gestionan el acceso a los datos.

En el proyecto se ha definido un paquete de java *model* que contiene la clase anterior y un subpaquete *chesspieces* donde están definidas todas las piezas disponibles. Para la implementación de las piezas se ha utilizado el concepto de *Herencia* [2] que se explica a continuación.

1) Herencia con las Piezas

Es un mecanismo muy utilizado en la programación orientada a objetos para poder reutilizar y extender código. Consiste en la habilidad de poder extender una clase conservando los métodos y atributos originales, de esta manera obtenemos una "*subclase*" que mantiene el comportamiento de la clase "*padre*".
En este caso se ha decidido utilizar este concepto debido a que las piezas del problema tienen un comportamiento muy parecido, en el que solo varia el movimiento de cada una. Este comportamiento se puede almacenar en una clase padre común que cada hija extiende.

Se ha definido la clase *Abstacta*[7] *Chesspiece* que contiene una lista de todas las posibles piezas y los métodos que cada una debe implementar

- *getType()*: este método debe retornar el tipo de pieza actual, se utiliza principalmente para la vista.
- *getDx() y getDy()*: estos métodos deben retornar una array con los vectores de movimiento de cada dimensión respectivamente.

Cada pieza implementada es una clase propia que extiende *Chesspiece* e implementa los métodos anteriores.

En la clase *Model* únicamente se utiliza el concepto de *Pieza* sin saber de cual se trata, como cada una implementa los métodos abstractos, se pueden invocar dichos métodos independientemente. De esta manera se puede computar una solución utilizando solo los movimientos de la pieza siendo esta anónima al modelo.

B. Implementacion de la Vista

Se ha diseñado la clase *View* que se encarga de mostrar al usuario toda la información del modelo, así como de recibir instrucciones para la ejecución.

Haciendo uso de componentes de la librería *Swing* se crea una interfaz intuitiva para el usuario, en primer lugar, se define un panel que muestra el tablero de ajedrez, donde cada casilla está formada por un *JLabel*.

En la parte inferior de la interfaz hay definidos elementos para dar información al usuario como un área de texto para informar del estado del programa y una barra de progreso que muestra el estado de la ejecución o animación.

Para finalizar, hay una serie de componentes que se encargan de recibir el *input* del usuario. En primer lugar, están los botones que se encargan de solucionar, animar o detener la ejecución. A continuación, hay un *JSpinner* para elegir la dimensión del tablero y un *JComboBox* para elegir la pieza con la que solucionar. Para elegir la posición inicial para el problema, basta con hacer *click* en la casilla deaseada del tablero.

[6] Un patrón de diseño es un conjunto de técnicas utilizadas para resolver problemas comunes en el desarrollo de software.

[7] Clase que declara existencia de métodos, pero no los implementa. Debido a esto una clase abstracta no puede ser instanciada.

C. *Implementacion del Controlador*

Se ha diseñado la clase *Controller* que se encarga de gestionar eventos y manejar la interacción entre el resto de las partes. Para ello se han implementado algunos métodos que servirán de *listeners* para los botones de la vista.

Cuando el usuario utilice algún botón de la interfaz este lo notifica al método correspondiente que realiza la acción necesaria si es posible y devuelve los resultados a la vista para que lo muestre. Para que el usuario pueda disfrutar de una interfaz gráfica fluida se hace uso del concepto de *Concurrencia* que se explica a continuación.

1) *Concurrencia*

La programación concurrente es una forma de cómputo en la que el trabajo se divide en varios hilos de ejecución distintos que trabajan simultáneamente. Esto suele mejorar el rendimiento de una aplicación al poder realizar cálculos largos en un hilo aparte.
Para este proyecto se ha decidido utilizar este concepto en dos apartados distintos:

- *Thread*
  Para poder computar el algoritmo recursivo sin dejar a la vista bloqueada, este cálculo se ejecuta en hilo diferente. Para ello se define el hilo usando la clase *Thread* de java con un elemento que implemente la interfaz *Runnable* el cual indica al hilo las acciones que debe realizar. En el método *run()* se obtienen los datos introducidos por el usuario en la vista y se computa la solución utilizando el modelo.

- *Animator*
  Para poder animar la solución en la vista sin dejarla bloqueada se ejecuta en un hilo distinto. Esto se consigue utilizando la clase *SwingWorker* diseñada especialmente para computaciones largas que utilicen componentes *Swing*,

Por último, cabe destacar que hay definida una clase adicional llamada *Message* que contiene una serie de *Strings* donde se almacenan los mensajes por defecto para que puedan ser usados independientemente por la vista. Además, hay definida una enumeración que contiene los distintos tipos de mensajes.

- Informativo
- Error
- Resultado

VI. GUIA DE USUARIO PARA LA APLICACIÓN

La primera imagen que se tiene de la interfaz una vez se ejecuta es la siguiente:

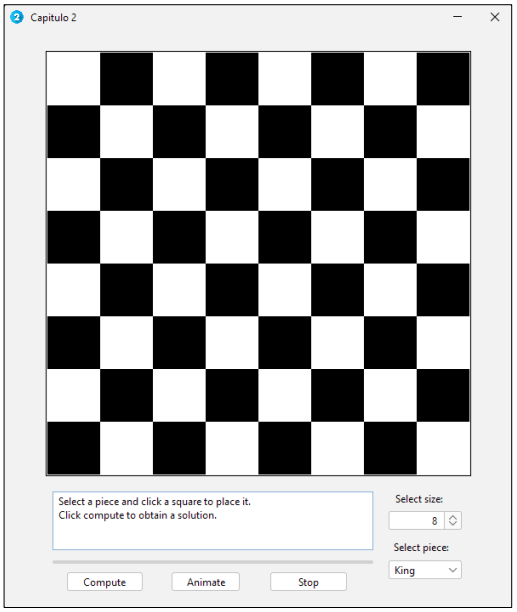

Ilustración 5: Vista inicial de la GIU

Se pueden distinguir dos partes: el tablero de ajedrez, y los botones de interacción juntamente con un cuadro de diálogo para la muestra de información y una barra de progreso.

Los pasos por seguir para realizar una ejecución son los siguientes:

1. Elegir el tamaño deseado. Se recomienda entre 5 y 8

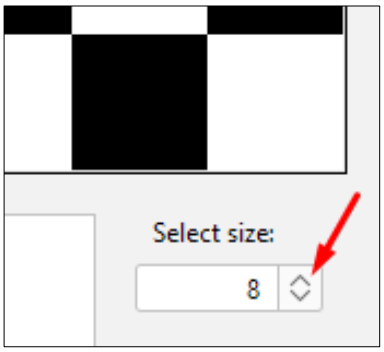

Ilustración 6: Elección del tamaño del tablero

2. Elegir la pieza deseada. Se puede elegir de entre cuatro piezas clásicas y tres piezas personalizadas.

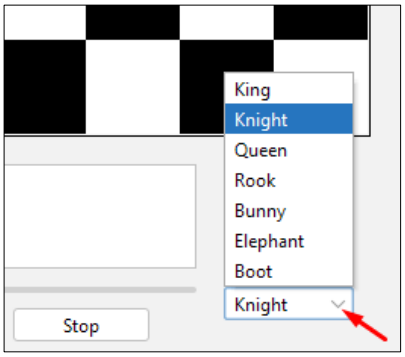

Ilustración 7: Elección de pieza

3. Hacer clic en la casilla del tablero desde donde se quiera empezar el camino.

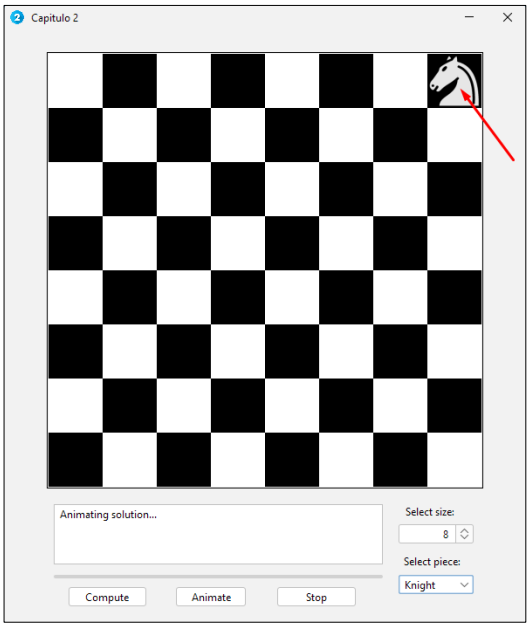

Ilustración 8: elección de casilla inicial

4. Dar clic en "Compute".
5. Una vez se haya encontrado la solución, si lo desea, puede darle a "*Animate*" y se mostrará una animación del recorrido, y la barra de progreso se irá actualizando.

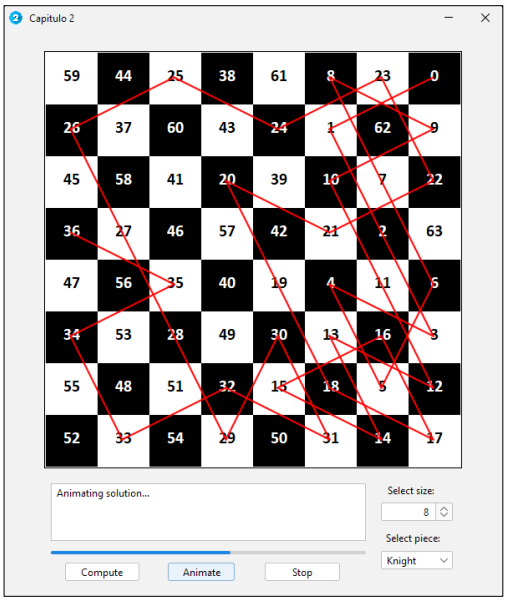

Ilustración 9: Animación del camino en progreso

VII. ESTUDIO DE RENDIMIENTO

Se ha hecho un pequeño estudio para intentar calcular la constante multiplicativa de un ordenador de uno de los integrantes del grupo.
Para hacerlo, se necesitan saber los siguientes datos:

- La complejidad asintótica del algoritmo
- El tiempo de ejecución

Una vez se tienen estos dos datos, se puede aplicar la siguiente fórmula:

$$t = CM * m^{N^2}$$

Ecuación 3

Para hacer las pruebas se ha elegido el caballo, con $m = 8$. Así que la ecuación que se tiene que resolver es la siguiente:

$$t = CM * 8^{N^2}$$

Ecuación 4

Aislando la Constante Multiplicativa, nos queda la siguiente fórmula:

$$CM = \frac{t}{8^{N^2}}$$

Ecuación 5

Teniendo esto en cuenta, hemos hecho unas cuantas pruebas en un par de tamaños y en distintas casillas, con los siguientes resultados:

Dimensión	**5**	**6**	**7**
Ejecución 1	0,00021	0,005969	0,145436
Ejecución 2	0,000535	110,5493	600
Ejecución 3	0,001188	0,000303	600
Ejecución 4	0,000223	0,026443	600
Ejecución 5	-	32,69588	-
Ejecución 6	-	0,001703	-
Promedio	0,000539	23,87994	450,0364
Aprox. CM	1,42672E-26	7,36E-32	2,52E-42

Tabla 1

Para calcular la constante multiplicativa hemos usado la Ecuación 3 antes mostrada, con el tiempo medio de un par de ejecuciones, y la dimensión usada para estas.
Cabe recalcar que para la dimensión 7, los 3 valores "600" se han puesto porque, después de 10 minutos, el programa aún no había encontrado solución, así que se ha puesto ese valor para rellenar, aunque debería ser más grande.

A. *Comparación de dimensiones*

En este estudio se busca comparar como aumenta el tiempo de ejecución para distintos tamaños de tablero. A su vez, se ha buscado realizar una comparativa entre distintas zonas del tablero. En particular, las posiciones de abajo a la izquierda y en medio. Cuando el tablero tenga una dimensión impar está claro qué casilla se considera la de en medio, en cambio, cuando la dimensión es par se ha elegido la casilla inferior izquierda de las cuatro centrales.

Dimensión 5	Inferior izquierda	Medio
Ejecución 1	0.000002	0.000196
Ejecución 2	0.000003	0.000195
Ejecución 3	0.000002	0.000318
Ejecución 4	0.000002	0.000192
Ejecución 5	0.000003	0.000197
Ejecución 6	0.000005	0.000322
Promedio	2.83333E-06	0.000236667
Aprox CM	7.49977E-29	6.26451E-27

Tabla 2

Dimensión 6	Inferior izquierda	Medio
Ejecución 1	0.000115	0.001881
Ejecución 2	0.000131	0.002104
Ejecución 3	0.000188	0.001616
Ejecución 4	0.000183	0.002006
Ejecución 5	0.000114	0.001649
Ejecución 6	0.000116	0.001622
Promedio	0.000141167	0.001813
Aprox CM	4.35003E-37	5.58674E-36

Tabla 3

Dimensión 7	Inferior izquierda	Medio
Ejecución 1	0.226195	0.054351
Ejecución 2	0.22603	0.054768
Ejecución 3	0.228409	0.056248
Ejecución 4	0.231506	0.05444
Ejecución 5	0.230465	0.057453
Ejecución 6	0.22455	0.054598
Promedio	0.227859167	0.055309667
Aprox CM	1.27719E-45	3.10021E-46

Tabla 4

B. Comparacion entre Casillas

Se ha realizado un estudio exhaustivo sobre el tiempo de ejecución de la solución en todas las casillas de un tablero *6x6*. Para cada casilla se ha computado la solución 5 veces y se ha realizado la media del tiempo de ejecución.
A continuación, se muestra una representación grafica

0.009	0.0734	0.56	31.66	0.0128	0.0024
144.22	0.0373	0.0805	0.0264	0.1873	0.0124
0.00079	42.58	0.0692	0.0145	0.0359	0.0177
0.3591	2.5	0.0034	0.0181	0.0078	0.2356
4.38	0.000096	0.0352	1.1945	13.88	0.00029
0.00043	0.0014	0.0043	0.004	0.0586	0.0064

Ilustración 10 Tiempo de ejecución en Segundos de cada casilla

Como se puede observar hay subidas muy drásticas para algunas casillas en concreto casualmente la mayoría de estas son de color negro. Se puede comprobar con las medias que en efecto las negras tienen un valor bastante superior, aunque esto es debido sobretodo a la casilla *(1,0)*.

Media Blancas:	1,21311901
Media Negras:	12,2491337

Tabla 5

Esto podría depender de varios aspectos como por ejemplo el orden en el que estén definidos los movimientos de la pieza, ya que puede dar preferencia a algunas combinaciones menos costosas.

VIII. CONCLUSIONES

A. Estudio de rendimiento

Como se puede observar, la constante multiplicativa es muy pequeña. Esto es debido a que casi todo el peso recae sobre la función asintótica, lo que no debería ser sorpresa al ser esta exponencial cuadrática.

Podemos observar el crecimiento según va creciendo el tamaño del tablero. Para N=5, apenas tarda un milisegundo en encontrar una solución. Para N=6, hay algunos en el orden de centésimas de segundos, pero otros en el orden de los 30 segundos o dos minutos. Para N=7, ya se empieza a complicar. La primera ejecución ha sido suertuda, de apenas 0,1 segundo, mientras que las otras se han alargado más de 10 minutos.

Esto se puede explicar con la función de complejidad asintótica. Al ser exponencial cuadrática, la velocidad en la que crece es inmensa. Para encontrar soluciones al tablero 8x8 tarda mucho, si pasamos al 9x9, aumentamos en 13 órdenes de magnitud.

B. Comparacion entre dimensiones

A continuación, se compararán los distintos tiempos de ejecución sobre una misma casilla en distintos tamaños de

tablero. Para ello se ha hecho la media de un total de 6 ejecuciones y calculado la media. Además, para obtener unos resultados más fiables, se han omitido las primeras tres ejecuciones en cada uno de los casos. Así de alguna manera reducimos el impacto que tiene la memoria caché al empezar a cargar los datos que más utiliza y, por ende, suele reducir el tiempo de ejecución.

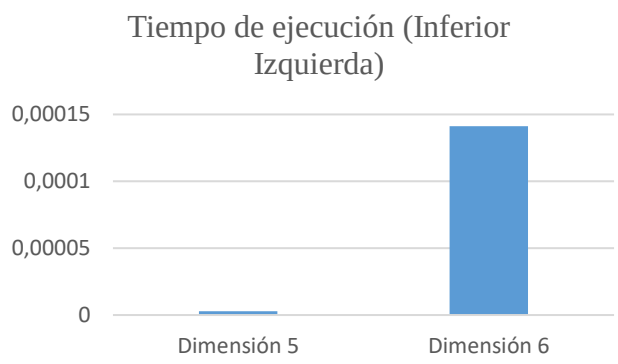

Ilustración 11 Gráfico de tiempos de ejecución

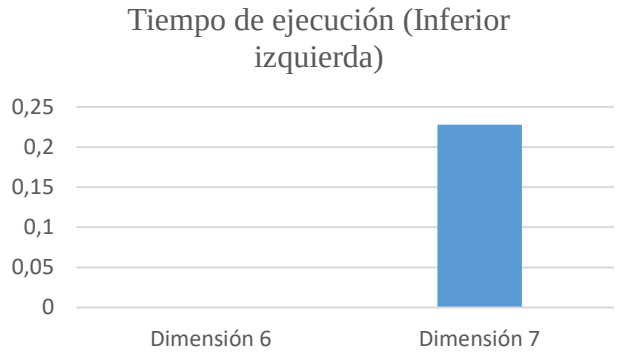

Ilustración 12 Gráfico de tiempos de ejecución

En primer lugar, es importante hacer la salvedad que no se han puesto las tres dimensiones en una misma gráfica porque la magnitud del tablero de 7x7 es tan grande que no permitiría poder comparar el resto de los tableros.
Ya con el simple hecho de tener que dividir las gráficas, deja entrever que los tiempos de ejecución se disparan. Comparando los datos, la diferencia de tiempos entre el tablero cinco y el seis es de aproximadamente 50 veces más tiempo en el de mayor tamaño. En porcentaje es alrededor de un 4900%.

Si por ejemplo atendemos al segundo gráfico, la diferencia es aún mayor, los tiempos aumentan un 161300 %. Tampoco tendría sentido comparar el tablero 5x5 con el 7x7, pero ya se puede observar que la diferencia sería abismal.

Esto deja a entrever uno de los grandes problemas del backtracking, un pequeño aumento en la dimensión del tablero hace que los costes temporales se disparen. Por eso es muy importante comprender la dificultad del problema que se quiere analizar y hasta qué punto es viable utilizar este tipo de algoritmos

C. *Es el backtracking factible para problemas grandes?*

Como se ha podido observar por la naturaleza de estos problemas, las formas de resolverlos son muy complicados computacionalmente. Se tienen que usar técnicas mejores que solo el backtracking (más optimizaciones, más podas del árbol recursivo, guardar soluciones parciales para las versiones simétricas de las mismas situaciones…) para poder resolver estos tipos de problemas cuando los tamaños sean suficientemente grandes.

D. *Necesidad del Modelo-Vista-Controlador*

Esta práctica es un buen ejemplo de la necesidad de una estructura como el Modelo-Vista-Controlador.
Al tener un proceso (el cálculo de los caminos) que está mucho tiempo ejecutándose, si no se tuviese la vista ejecutándose en un hilo distinto, el programa se quedaría bloqueado.
Además, se ha podido aprovechar la estructura del primer capítulo, al ser el MVC una estructura fácilmente reutilizable.

IX. Referencias

[1] Modelo–vista–controlador - Wikipedia, la enciclopedia libre.
[2] Herencia (informática) - Wikipedia, la enciclopedia libre.
[3] Ciclo (Hamiltoniano) – Wikipedia, la enciclopedia libre.
[4] Chess Attitude – Movimiento de las piezas
[5] Techopedia - Overflow

X. Bibliografía

[1] [En línea]. Available: https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador.

[2] [En línea].

[3] «Wikipedia,» [En línea]. Available: https://hmong.es/wiki/Hamiltonian_cycle.

[4] [En línea]. Available: https://hmong.es/wiki/Hamiltonian_cycle.

[5] Wikipedia. [En línea]. Available: https://hmong.es/wiki/Hamiltonian_cycle.

[6] «ChessAttitude,» [En línea]. Available: https://www.chessattitude.com/?page_id=278#:~:text=El%20movimiento%20del%20caballo%20es,parece%20a%20letra%20%22L%22..

[7] Wikipedia, «Backtracking - Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/Backtracking.