Universidad Zaragoza
1542

Trabajo Fin de Grado

Desarrollo de un juego de ajedrez con inteligencia artificial basada en el algoritmo Minimax

Development of a Chess Game with Minimax algorithm-based artificial intelligence

Autor

David Gispert Gutiérrez

Director

José Javier Merseguer Hernáiz

ESCUELA DE INGENIERÍA Y ARQUITECTURA
Septiembre 2024

Índice

Capítulo 1

Introducción

El desarrollo de un juego de ajedrez digital requiere una comprensión profunda tanto de las reglas y la mecánica del juego [1] como de los principios de diseño de software [2]. Este TFG aborda varios aspectos críticos del proceso, incluyendo la representación de la interfaz de usuario, la implementación de las reglas del juego, y la creación de una inteligencia artificial [3] capaz de desafiar a los jugadores.

El estado del arte en las aplicaciones de ajedrez ha sido significativamente impulsado por desarrollos como Stockfish [4] y AlphaZero [5, 6]. Stockfish es uno de los motores de ajedrez más poderosos, conocido por su capacidad de búsqueda y evaluación eficiente, y es utilizado ampliamente tanto en competiciones como en análisis. Por otro lado, AlphaZero, desarrollado por DeepMind[1], ha demostrado capacidades sorprendentes al aprender a jugar ajedrez mediante técnicas de aprendizaje profundo y autoaprendizaje, logrando superar a los mejores motores tradicionales sin conocimiento previo del juego. Estas aplicaciones no sólo han elevado el nivel de competencia en el ajedrez digital, sino que también han marcado hitos en el campo de la inteligencia artificial.

En este contexto, nuestro proyecto se posiciona como un intento de seguir el camino de los motores tradicionales, centrándose en técnicas clásicas de búsqueda y evaluación en el ambiente de un marco educativo. No pretendemos competir directamente con los gigantes mencionados, sino proporcionar una plataforma accesible y didáctica que permita a los jugadores y desarrolladores comprender los fundamentos del ajedrez digital y la inteligencia artificial aplicada al mismo.

Para llevar a cabo el desarrollo de este proyecto, utilizaremos C# [7, 8] como lenguaje de programación y el motor de videojuegos Godot [9]. Godot es una herramienta de desarrollo de juegos de código abierto que ofrece un entorno integrado y completo para crear videojuegos 2D y 3D. Es conocido por su flexibilidad, facilidad de uso y una comunidad activa que contribuye a su constante mejora. El uso de Godot nos permitirá desarrollar tanto la lógica del juego como la interfaz de usuario de manera

[1] https://deepmind.google

eficiente y cohesiva.

Este trabajo no sólo pretende crear un juego de ajedrez funcional y atractivo, sino también servir como una guía detallada y educativa para aquellos interesados en el desarrollo de videojuegos, demostrando cómo se puede abordar un proyecto complejo de manera sistemática y efectiva. Además, este TFG examinará los desafíos encontrados durante el proceso de desarrollo y las soluciones implementadas para superarlos.Cada etapa del desarrollo presenta sus propios obstáculos y oportunidades de aprendizaje.

Los Capítulos 2 y 3 describen la estructura fundamental del juego, que abarca desde la implementación del tablero y las piezas hasta los mecanismos de control y la lógica subyacente que rige las interacciones del juego. El Capítulo 4 describe cómo se ha desarrollado la interfaz de usuario. El Capítulo 5 explica la inteligencia artificial desarrollada para el juego. Finalmente, el Capítulo 6 presentará los resultados obtenidos, destacando las características principales del juego desarrollado y evaluando su desempeño tanto desde el punto de vista técnico como desde la experiencia del usuario. Concluiremos con una reflexión sobre las posibles mejoras futuras y las implicaciones del proyecto en el ámbito del desarrollo de videojuegos y la educación en programación.

El código fuente del proyecto está disponible en el siguiente enlace:

`https://github.com/Flyboy1010/ChessTFG`.

Capítulo 2

Desarrollo de la estructura del juego

Cualquier sistema software complejo, y los videojuegos sin ninguna duda lo son, tanto más un videojuego de ajedrez, queda caracterizado por su parte *estática* y su parte *dinámica* [10]. La parte *estática* de un sistema identifica la información que subyace en el mismo, en términos de sus componentes principales o *clases*. A su vez, la parte *dinámica* del sistema comprende tanto la definición de las reglas del negocio como las interacciones que se producen entre los componentes estáticos previamente identificados.

En el desarrollo de un juego de ajedrez la parte *estática* requiere por tanto la creación de varios componentes esenciales, o clases, que den soporte a la estructura del juego. En este capítulo, desarrollamos los componentes que servirán de base para el resto del programa. Estos componentes son las piezas, el tablero, los movimientos y los jugadores. A continuación, describimos cada uno de estos elementos. Además, a la hora de describir cada clase hacemos hincapié en cómo cada una de las **decisiones de diseño** que hemos tomado repercute en la implementación del sistema. En la Sección 2.6 recapitularemos para ofrecer una visión de conjunto de dicha estructura, en términos de un diagrama de clases [11].

Por otro lado, en el juego del ajedrez la parte *dinámica* queda obviamente definida por las reglas del juego, que suponen a su vez las interacciones entre los componentes identificados en la parte estática. El Capítulo 3 presentará esta parte del diseño software.

2.1. Las piezas

Las piezas de ajedrez son los componentes fundamentales que se desplazan por el tablero. Cada pieza se define por dos parámetros: su tipo y su color.

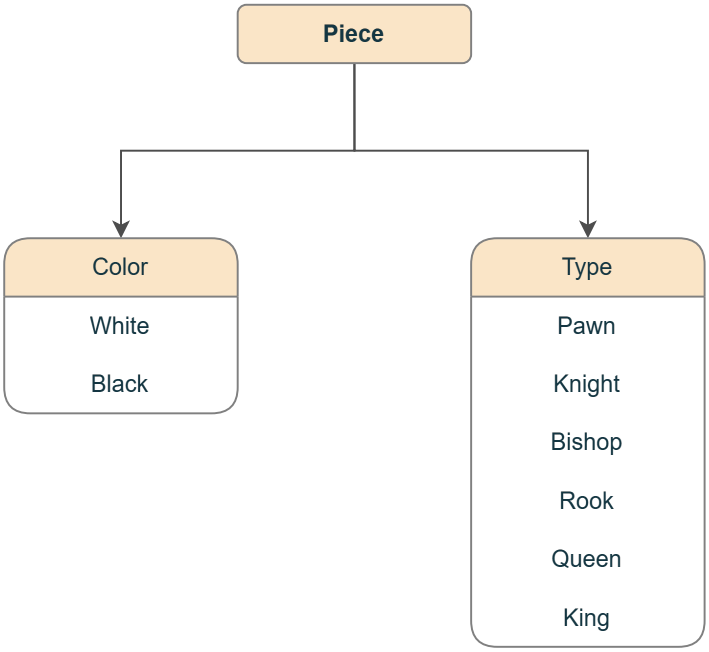

Figura 2.1: Estructura de una pieza de ajedrez

Implementacion

La clase *Piece* está diseñada para representar las piezas en un juego de ajedrez. La implementación resultante es la siguiente:

```
1  public struct Piece
2  {
3      public enum Type
4      {
5          None,
6          King,
7          Queen,
8          Bishop,
9          Knight,
10         Rook,
11         Pawn
12     }
13
14     public enum Color
15     {
16         None,
17         White,
18         Black
19     }
20
21     public Type type;
22     public Color color;
23 }
```

Fragmento de código 2.1: Clase *Piece* simplificada

Para almacenar los valores color y tipo hemos decido utilizar 2 enumeraciones con los valores mostrados en la figura 2.1.

2.2. Los movimientos

Los movimientos determinan la forma en que las piezas se mueven a través del tablero. Un movimiento se define como el traslado de una pieza desde una casilla de origen A hasta una casilla de destino B. Para realizar un movimiento, es necesario conocer estas dos casillas. Con esta información, podríamos crear una estructura que almacene estos dos parámetros únicos, definiendo así un movimiento. No obstante, por razones que explicaremos más adelante, incluiremos también el tipo de la pieza de origen y el tipo de la pieza de destino, siendo esta última una pieza de tipo *vacía* si no hay ninguna en la casilla de destino.

Además, debido a que existen movimientos especiales como el enroque o la captura al paso, añadiremos una variable que especifique si el movimiento corresponde a uno de estos casos especiales.

Implementación

La clase *Move* encapsula todos los parámetros que hemos mencionado anteriormente. Aquí podemos ver el codigo:

```
1  public struct Move
2  {
3      // move flags
4
5      public enum Flags
6      {
7          None,
8          DoublePush,
9          Promotion,
10         EnPassant,
11         CastleShort,
12         CastleLong
13     }
14
15     public int squareSourceIndex, squareTargetIndex;
16     public Piece pieceSource, pieceTarget;
17     public Flags flags;
18
19     public Piece.Type promotionPieceType;
20 }
```

Fragmento de código 2.2: Clase *Move* simplificada

A continuación se presenta una explicación detallada de su funcionalidad y estructura:

Enumeración *Flags*

La enumeración *Flags* define varios tipos de movimientos especiales en el ajedrez:

- *None*: Movimiento normal sin características especiales.
- *DoublePush*: Movimiento inicial del peón que avanza dos casillas.
- *Promotion*: Indica que el movimiento es una promoción de un peón.
- *EnPassant*: Indica que el movimiento realiza una captura al paso.
- *CastleShort*: Indica que el movimiento es un enroque corto.
- *CastleLong*: Indica que el movimiento es un enroque largo.

Atributos de la Clase

La estructura *Move* contiene varios atributos que describen un movimiento:

- *squareSourceIndex*: Índice de la casilla de origen.
- *squareTargetIndex*: Índice de la casilla de destino.
- *pieceSource*: Pieza que se mueve.
- *pieceTarget*: Pieza en la casilla de destino (puede ser nula si la casilla está vacía).
- *flags*: Indica el flag del movimiento.
- *promotionPieceType*: Tipo de pieza en la que se convierte un peón al promocionarse.

2.3. El tablero

El tablero de ajedrez es una cuadrícula de 8x8, formada por 64 casillas alternadas en colores claros y oscuros. Cada casilla puede estar vacía o contener una pieza.

Implementación

Podemos entender el tablero como una interfaz para realizar movimientos. Este llevará un seguimiento de las piezas y de variables de control que indicarán las acciones disponibles en el estado actual, y expondrá una serie de métodos que nos permitirán interactuar con él indicándole el movimiento que deseamos realizar.

Una de las primeras aproximaciones para gestionar la estructura del tablero es crear una clase que almacene información esencial, como las piezas, el color del turno

actual y la posibilidad de realizar enroques, entre otros. Sin embargo, pronto se hace evidente que esta solución no es óptima debido a que ciertas acciones dependen del estado anterior del tablero. Un ejemplo claro es la captura al paso, que solo puede realizarse si el oponente ha movido un peón dos casillas en el movimiento anterior.

Para abordar este problema, se propone dividir la clase del tablero en dos componentes: *BoardState* y *Board*.

Clase BoardState

La clase *BoardState* contendrá todos los indicadores del estado del juego, tales como el turno actual, la posibilidad de enroque y la disponibilidad de captura al paso. Esta clase se encarga de gestionar y almacenar la información relativa al estado del juego en un momento dado.

```
1  public struct BoardState
2  {
3      private Piece.Color turnColor = Piece.Color.None;
4
5      /* EN PASSANT */
6      private bool isEnPassantAvailable = false;
7
8      private Piece.Color doublePushedPawnColor = Piece.Color.None;
9
10     private int enPassantSquareIndex = -1;
11
12     /* CASTLELING */
13     private bool canCastleShortWhite = false;
14     private bool canCastleLongWhite = false;
15
16     private bool canCastleShortBlack = false;
17     private bool canCastleLongBlack = false;
18
19     // half move count
20     private int halfMoveCount = 0;
21 }
```

Fragmento de código 2.3: Clase *BoardState* simplificada

Los indicadores a tener en cuenta son los siguientes:

- **Turno actual**: Flags relacionados con el turno actual
  - *turnColor*: Indica el color del jugador cuyo turno es el actual.
- **Captura al paso**: Flags relacionados con la captura al paso
  - *isEnPassantAvailable*: Indica si la captura al paso (en passant) está disponible.

- *doublePushedPawnColor*: Almacena el color del peón que realizó un movimiento doble en el turno anterior, utilizado para verificar si la captura al paso es posible.
- *enPassantSquareIndex*: Guarda la posición del cuadrado del peón que puede ser capturado mediante la captura al paso.

– **Enroque**: Flags relacionados con el enroque

- *canCastleShort*: Indica si se puede realizar el enroque corto.
- *canCastleLong*: Indica si se puede realizar el enroque largo.

– **Conteo de movimientos**: Flags relacionados con el enroque

- *halfMoveCount*: Indica el numero de medios movimientos que se han realizado a lo largo de la partida.

Clase Board

Por otro lado, la clase *Board* actuará como el contenedor principal que no solo llevará el seguimiento de las piezas en el tablero, sino que también mantendrá un registro tanto del estado actual como de los estados anteriores del juego. Esta separación es esencial porque, para ciertos movimientos como la captura al paso o el enroque, es necesario conocer el estado del tablero en jugadas anteriores.

Además, la clase Board se comporta como una interfaz que expone funciones para realizar movimientos.

Aquí podemos ver una simplificación de su implementación en código.

```
1  public class Board
2  {
3      private Piece[] pieces = new Piece[64];
4
5      private Stack<BoardState> boardStates = new Stack<BoardState>();
6
7      private BoardState currentBoardState = new BoardState();
8
9      public Piece GetPiece(int index)
10     {
11         return pieces[index];
12     }
13
14     private void SetPiece(int index, Piece piece)
15     {
16         pieces[index] = piece;
17     }
18
19     public void MakeMove(Move move)
20     {
```

```
21          // save previous board state
22
23          BoardState previousBoardState = currentBoardState;
24
25          // save current board state
26
27          boardStates.Push(currentBoardState);
28
29          // check move flags
30
31          switch (move.flags)
32          {
33              ... // handle special moves
34              default:
35                  // move the piece to the target
36
37                  SetPiece(move.squareSourceIndex, Piece.NullPiece);
38                  SetPiece(move.squareTargetIndex, move.pieceSource);
39                  break;
40          }
41
42          // change turn color
43
44          Piece.Color turnColor = currentBoardState.GetTurnColor();
45          currentBoardState.SetTurnColor(Piece.GetOppositeColor(turnColor));
46      }
47  }
```

Fragmento de código 2.4: Clase *Board* simplificada

Las piezas se almacenan en un array de dimension 64, el cual contendrá las piezas que hay actualmente en el tablero y el cual se irá modificando a la vez que se vayan efectuando movimientos.

Para poder realizar movimientos disponemos de una función *MakeMove*, la cual toma como parámetro una estructura del tipo *Move*. Esta función se encargará de modificar el array de piezas y el estado del tablero en base a las acciones definidas en la estructura *Move*. Primero, se guarda el estado actual del tablero en una pila por lo que hemos mencionado que para ciertos movimientos necesitamos conocer el estado anterior. A continuación, según los indicadores del movimiento (*flags*), la función realizará las acciones correspondientes, como mover una pieza de una casilla a otra, capturar una pieza, o manejar movimientos especiales como enroques o promociones.

Veamos un ejemplo en el caso en el que un peón se mueve dos casillas:

```
1  case Move.Flags.DoublePush:
2      // enable en passant flags in board state
3
4      currentBoardState.SetEnPassant(true);
5      currentBoardState.SetEnPassantColor(move.pieceSource.color);
6      currentBoardState.SetEnPassantSquareIndex(move.squareTargetIndex);
7
```

```
8       // move the piece from source to target
9
10      SetPiece(move.squareSourceIndex, Piece.NullPiece);
11      SetPiece(move.squareTargetIndex, move.pieceSource);
12      break;
```

Fragmento de código 2.5: Caso peon se mueve 2 casillas

Como se ha movido un peón dos casillas, eso significa que ese peón se puede capturar mediante una captura al paso en el turno siguiente, por lo que tendremos que modificar el estado del tablero indicando que una captura al paso es posible, el color del peón y el índice del cuadrado donde es posible realizar la captura.

A continuación se mueve como en el resto de movimientos la pieza origen, en este caso el peón hacia la casilla destino, y se coloca una pieza *vacía* en el lugar en el que se encontraba previamente.

Con el resto de movimientos especiales se sigue un procedimiento similar.

Finalmente, la función cambia el color del turno para reflejar que se ha completado el movimiento.

2.4. Los jugadores

El ajedrez involucra dos jugadores, cada uno controlando un conjunto de piezas (blancas o negras). Los jugadores pueden ser de dos tipos:

- *Humanos*: Introducen movimientos manualmente a través de la interfaz de usuario.
- *Inteligencia Artificial (IA)*: Calcula los movimientos automáticamente utilizando algoritmos de búsqueda y evaluación.

Implementación

La clase *Player* la proponemos como una clase abstracta, que define el comportamiento general de un jugador en el juego de ajedrez. Esta clase no puede ser instanciada directamente, sino que sirve como base para clases derivadas que implementan jugadores específicos, como un jugador humano o una inteligencia artificial.

Aquí podemos ver la implementación:

```
1 public abstract class Player
2 {
3     public event System.Action<Move> onMoveChosen;
4
```

```
5       public abstract void Update();
6
7       public abstract void NotifyTurnToMove();
8
9       protected virtual void ChoseMove(Move move)
10      {
11          onMoveChosen?.Invoke(move);
12      }
13  }
```

Fragmento de código 2.6: Clase *Player*

Método *Update*

Este método debe ser implementado por las clases derivadas y se utiliza para actualizar el estado del jugador en cada iteración del *game loop*. Por ejemplo, para el caso de un jugador humano, este método manejará la entrada del usuario para seleccionar el movimiento que se quiere realizar.

Método *NotifyTurnToMove*

Este método debe ser implementado por las clases derivadas y se llama cuando es el turno del jugador para mover. Permite que el jugador prepare y ejecute su movimiento.

Método *ChoseMove* y evento *onMoveChosen*

El método *ChoseMove* será llamado por el propio jugador cuando éste haya seleccionado un movimiento. Este método, a su vez, invocará el evento *onMoveChosen*, pasando como parámetro el movimiento elegido. El *game loop*, que actúa como el pegamento del sistema, estará suscrito a este evento, permitiendo la integración y coordinación de las diferentes partes del juego. Mas adelante en la sección 2.5, cuando hablemos en profundidad acerca del *game loop* se entenderá el porqué de este sistema.

Ahora nos tocaría implementar el jugador humano y el jugador controlado por la *IA*, pero ambos dependen de la implementación de otras partes del juego. En el caso del jugador humano, éste depende de la interfaz de usuario, debido a que tiene que ser capaz de controlar las piezas para poder realizar movimientos. Y en el caso del jugador controlado por la *IA*, necesitamos que la *IA* esté implementada. Por estos motivos, los veremos mas adelante.

2.5. La clase *Game*

La clase *Game* es el elemento central en el desarrollo de nuestro videojuego de ajedrez. Actuará como el núcleo que controla y coordina todos los aspectos del juego.

Esta clase se encargará de inicializar el tablero y los jugadores, además de contener el *game loop* el cual se regirá por una máquina de estados.

2.5.1. La máquina de estados

Para nuestro juego de ajedrez, hemos definido los siguientes estados:

- **PlayerTurn**: En este estado se actualizará el jugador al cual le toca mover, esto le permitirá ejecutar toda la lógica correspondiente a la selección del movimiento que quiere realizar. Una vez elegido el movimiento, se comprobará si el juego ha terminado y se cambiará al estado *NextTurn* si el juego no ha terminado y al estado *Over* si ha terminado.
- **NextTurn**: En este estado se notificará al jugador al cual le va tocar en este turno. Una vez notificado se cambia de estado a *PlayerTurn*.
- **Over**: El juego ha terminado.

Aquí podemos ver un esquema de la máquina de estados con sus transiciones:

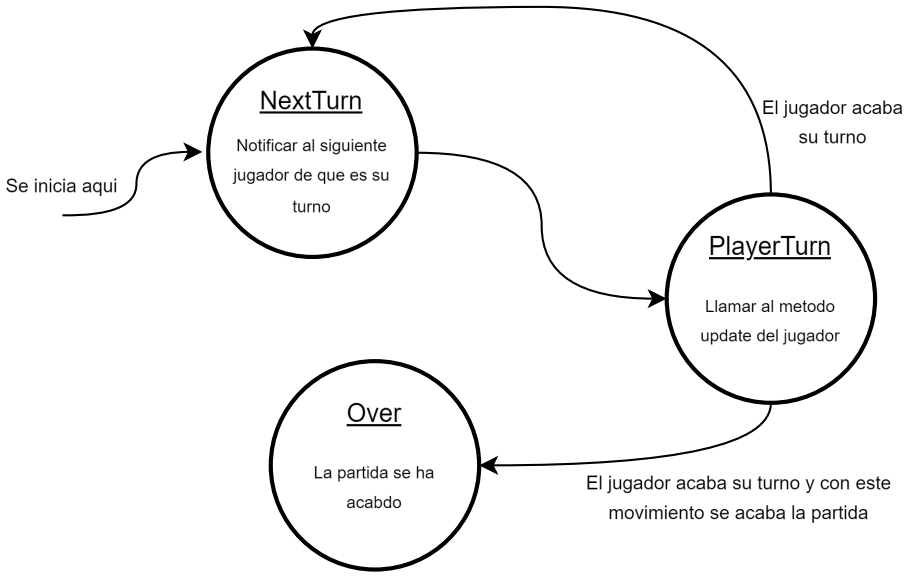

Figura 2.2: Máquina de estados del juego

2.5.2. Implementación

La implementación de la clase *Game* es la siguiente:

```
1 using Godot;
2 using System;
3
4 public partial class Game : Node
```

```
5 {
6     // game state machine
7
8     private enum GameState
9     {
10        PlayerTurn,
11        NextTurn,
12        Over
13    }
14
15    // board class that contains everything related to the pieces
16
17    private Board board;
18
19    // current game state
20
21    private GameState gameState;
22
23    // players
24
25    private Player playerWhite;
26    private Player playerBlack;
27
28    // current player
29
30    private Player playerToMove;
31
32    // Called when the node enters the scene tree for the first time.
33
34    public override void _Ready() {...}
35
36    // on move chosen
37
38    private void OnMoveChosen(Move move) {...}
39
40    // update game state
41
42    private void UpdateState() {...}
43
44    // Called every frame. 'delta' is the elapsed time since the previous frame.
45
46    public override void _Process(double delta) {...}
47 }
```

Fragmento de código 2.7: Clase *Game* simplificada

La clase *Game* hereda de la clase *Node* proporcionada por Godot. En Godot, todo se construye a partir de nodos. Un nodo es una unidad básica que puede realizar una función específica, como representar una imagen, manejar entradas del usuario, ejecutar scripts, o gestionar la física. Los nodos se organizan en una estructura de árbol, donde cada nodo puede tener múltiples nodos hijos. Esta jerarquía facilita la organización y el control de los elementos del juego, permitiendo una flexibilidad significativa a la hora de diseñar y programar. En este caso el nodo *Node*[12] nos da acceso a 2 metodos importantes, el metodo *_Ready* y el metodo *_Process*.

A continuación explicaremos las diferentes partes de la clase *Game*.

Inicialización

En Godot, el método *_Ready* se utiliza para inicializar [12]. Dentro de este método, configuramos el tablero y creamos los jugadores, preparando así el juego para su comienzo.

```
public override void _Ready()
{
    // inicializamos el tablero
    board = new Board();
    board.LoadFenString(Board.StartFEN);

    // creamos los jugadores (asi es como quedaria)
    // playerWhite = new PlayerHuman(board);
    // playerBlack = new PlayerAI(board);

    // nos suscribimos a los eventos de ambos jugadores
    playerWhite.onMoveChosen += OnMoveChosen;
    playerBlack.onMoveChosen += OnMoveChosen;

    // comenzamos en el estado NextTurn
    gameState = GameState.NextTurn;
}
```

Fragmento de código 2.8: Inicialización

Tenemos que destacar que durante la inicialización, nos suscribimos a los eventos *onMoveChosen* de ambos jugadores, por lo que cuando estos realizen un movimiento se llame la función *OnMoveChosen* al que se le pasara el movimiento que estos hayan elegido.

Transiciones de Estados

El método *UpdateState* contiene la lógica para manejar los diferentes estados del juego y realizar las transiciones apropiadas con respecto a la maquina de estados previamente descrita en la figura 2.2.

```
private void UpdateState()
{
    switch (gameState)
    {
        case GameState.NextTurn:
            Piece.Color turnColor = board.GetTurnColor();
            playerToMove = turnColor == Piece.Color.White ? playerWhite :
            playerBlack;
            playerToMove.NotifyTurnToMove();
            gameState = GameState.PlayerTurn;
            break;
        case GameState.PlayerTurn:
```

```
13              playerToMove.Update();
14              break;
15          case GameState.Over:
16              break;
17      }
18  }
```

Fragmento de código 2.9: Función *UpdateState*

Podemos ver que cuando nos encontramos en el estado *PlayerTurn* lo único que hacemos es actualizar al jugador y puede parecer que de este estado no saldremos nunca. Sin embargo, cuando un jugador elige un movimiento en su método *Update* mediante la llamada al método *ChooseMove* de la clase *Player*. El cual como vimos en el apartado 2.4 invocaba el evento *onMoveChosen* al cual le pasabamos el movimiento elegido, provoca entonces que el método *OnMoveChosen* de la clase *Game* se llame debido a que previamente en la inicialización nos habíamos suscrito al evento *onMoveChosen* del jugador, aplicando el movimiento al tablero y cambiando el estado del juego a *NextTurn*.

```
1  private void OnMoveChosen(Move move)
2  {
3      board.MakeMove(move);
4      gameState = GameState.NextTurn;
5  }
```

Fragmento de código 2.10: Función *OnMoveChosen*

Procesamiento del estado del juego

En Godot, el método *_Process* se ejecuta cada *frame* [13], por lo que este método actuará como nuestro *game loop*. En este método llamaremos a *UpdateState* para que así se vayan actualizando los estados.

```
1  public override void _Process(double delta)
2  {
3      UpdateState();
4  }
```

Fragmento de código 2.11: El *Game Loop*

2.6. La estructura en su conjunto

El diagrama de clases que aparece en la Figura 2.3 representa la estructura del juego en su conjunto. A partir de este diagrama se han realizado las implementaciones previas. En él se pueden observar de forma mas general las relaciones que existen entre todos estos componentes que hemos ido explicando a lo largo de este capítulo.

Finalmente, con todos estos elementos básicos que hemos ido desarrollando podemos ya pasar a la implementación de las reglas del juego.

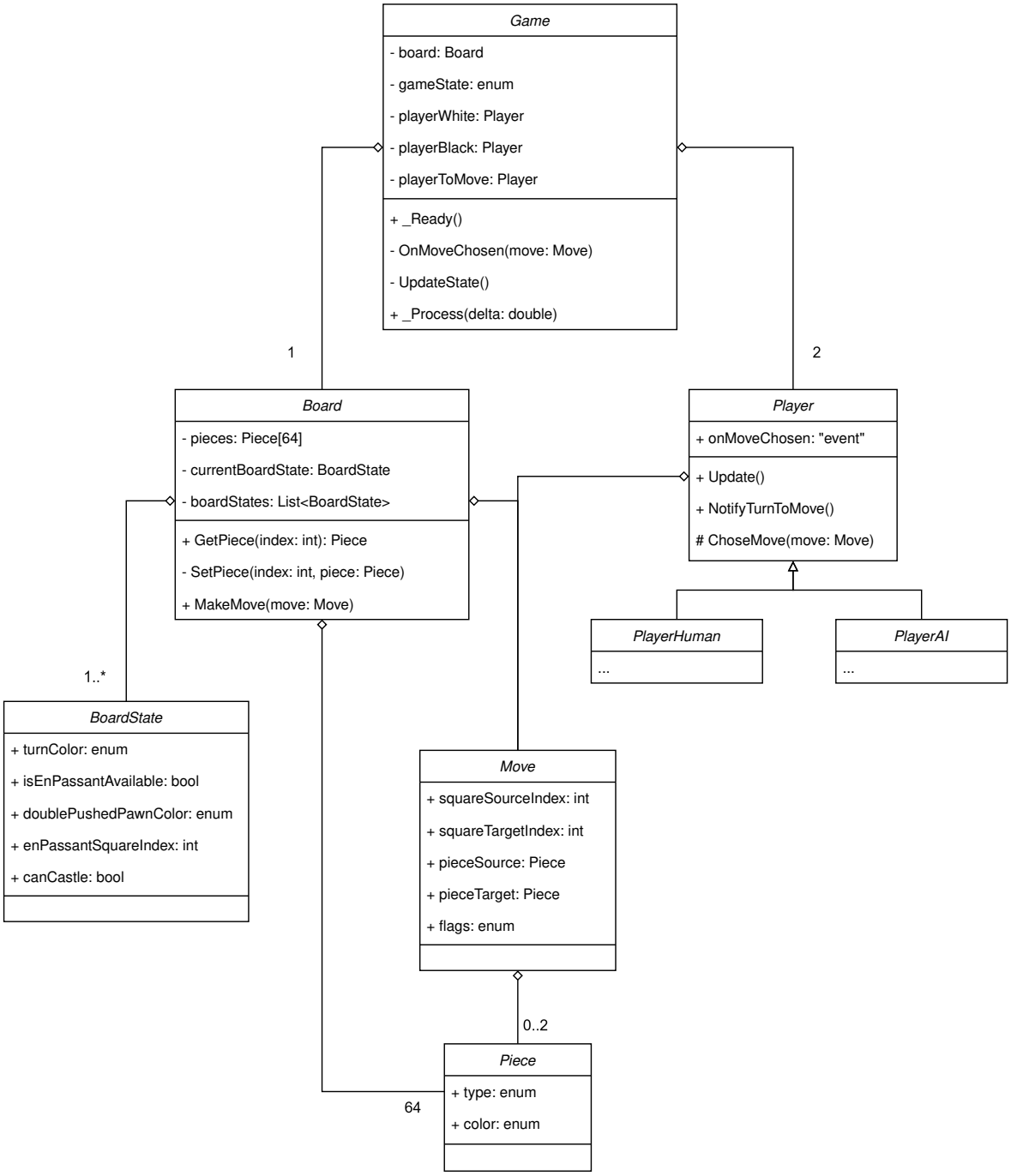

Figura 2.3: Diagrama de clases de la estructura básica del juego

Capítulo 3

Implementación de las reglas

Una vez definida la estructura básica del programa podemos ya implememtentar las reglas del ajedrez, es decir, la *dinámica* del sistema. Esto abarca desde las restricciones de movimiento de cada pieza, que se desarrollará en la Sección 3.1, hasta las comprobaciones de jaque y jaque mate, desarrolladas en la Sección 3.2. Por último, las tablas por triple repetición y la regla de los 50 movimientos se acometen en las Secciónes 3.3 y 3.4 respectivamente.

3.1. Movimientos de las piezas

En el juego del ajedrez cada tipo de pieza se mueve de una manera diferente. Además, hay ciertas piezas que tienen movimientos especiales que sólo se pueden realizar si se cumplen una serie de condiciones en el tablero y/o en estados anteriores del mismo.

Cabe resaltar que debemos intentar que la generación de los movimientos sea lo más rápida posible, puesto que más adelante cuando desarrollemos la *IA*, cuanto menos tiempo tarden en generarse los movimientos de cada pieza, más en profundidad se podrá analizar el juego y por lo tanto la *IA* será un mejor rival. Para ello, intentaremos en la medida de lo posible precalcular todos los movimientos posibles. Ahora veremos a qué nos referimos con esto.

3.1.1. El Caballo

Veamos un ejemplo de cómo se mueve el caballo. Si se encontrase en la casilla de la Figura 3.1, se podría mover a todas las casillas marcadas con un círculo verde. Como nos interesa precalcular todo lo posible, para que sea más rápido, lo que podemos hacer es para cada casilla del tablero ver a qué casillas se podría mover si se encontrase en dicha casilla.

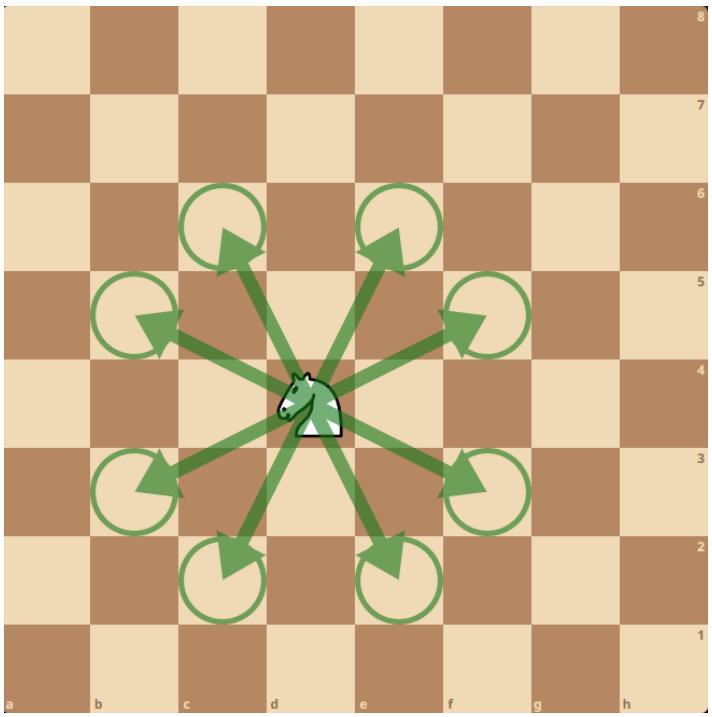

Figura 3.1: Movimiento del caballo

Para ello lo que haremos es recorrer las 64 casillas del tablero y guardarnos en una *LookUpTable* las casillas disponibles. Así, cuando queramos saber a qué casillas se podría mover si se encontrase en la casilla X, consultaríamos la tabla de la casilla X.

Aquí podemos ver el código que genera dichos movimientos:

```
1  int[][] preCalculatedKnightMoves = new int[64][];
2
3  List<int> movesBuffer = new List<int>();
4
5  for (int j = 0; j < 8; j++)
6  {
7      for (int i = 0; i < 8; i++)
8      {
9          int index = i + j * 8;
10
11         for (int jj = -2; jj <= 2; jj += 4)
12         {
13             for (int ii = -1; ii <= 1; ii += 2)
14             {
15                 if (IsInBounds(i + ii, j + jj))
16                 {
17                     movesBuffer.Add((i + ii) + (j + jj) * 8);
18                 }
19
20                 if (IsInBounds(i + jj, j + ii))
21                 {
22                     movesBuffer.Add((i + jj) + (j + ii) * 8);
23                 }
24             }
25         }
```

```
26
27          preCalculatedKnightMoves[index] = movesBuffer.ToArray();
28      }
29  }
```

Fragmento de código 3.1: Precalculación de los movimientos del caballo

El array *preCalculatedKnightMoves* es una *LookUpTable* que se usa para almacenar todos los movimientos posibles de un caballo en un tablero de ajedrez, para cada una de las 64 posiciones del tablero. Este es un array de arrays, donde cada elemento es un array que contiene los posibles movimientos desde una posición específica del tablero. El tamaño principal del array es 64, correspondiente a las 64 casillas. Dentro de los bucles anidados, se calculan las posibles posiciones de destino para los movimientos del caballo (que pueden ser 2 casillas en una dirección y 1 en otra, en todas las combinaciones posibles). Si volvemos a la Figura 3.1, esto correspondería a la generación de los círculos verdes.

Tenemos que resaltar que esto sólo nos devuelve las casillas disponibles, posteriormente deberemos comprobar si esas casillas se encuentran vacías, o con una pieza. En ese caso tendríamos que comprobar el color de la pieza y ver si corresponde al color contrario, entonces el movimiento sería valido, puesto que esta pieza podría ser capturada. Sin embargo, si el color de la pieza es el mismo que el del caballo, entonces no sería un movimiento válido.

Podemos ver aquí finalmente el código que genera el movimiento de un caballo, al cual le pasamos un tablero y la casilla donde se encuentra el caballo, y devuelve una lista de los movimientos posibles.

```
1  private static List<Move> GenerateKnightMoves(Board board, int index)
2  {
3      List<Move> moves = new List<Move>();
4
5      Piece piece = board.GetPiece(index);
6
7      foreach (int targetIndex in preCalculatedKnightMoves[index])
8      {
9          Piece targetPiece = board.GetPiece(targetIndex);
10
11         if (targetPiece.color == piece.color)
12             continue;
13
14         moves.Add(new Move()
15         {
16             squareSourceIndex = index,
17             squareTargetIndex = targetIndex,
18             pieceSource = piece,
19             pieceTarget = targetPiece
20         });
21     }
22
```

```
23      return moves;
24  }
```

Fragmento de código 3.2: Generación del movimiento de un caballo

Observamos que se hace uso de la *LookUpTable* *preCalculatedKnightMoves* del caballo para el índice de la casilla seleccionada, y para cada una de las casillas disponibles debemos comprobar, como hemos dicho previamente, que se encuentren vacías o con una pieza del color contrario para garantizar la validez del movimiento. Una vez comprobado que el movimiento es válido, construimos el movimiento de acuerdo a los parámetros de la Sección 2.2 y lo añadimos a la lista.

3.1.2. La Torre, el Alfil y la Reina

La torre, el álfil y la reina son piezas que se comportan prácticamente de la misma manera. Estas piezas se deslizan por el tablero hasta que se encuentran con el límite de este o con otra pieza. La torre se desliza de manera horizontal y vertical. El álfil se desliza a lo largo de las diagonales, y la reina es una combinación de ambos, vease la Figura 3.2.

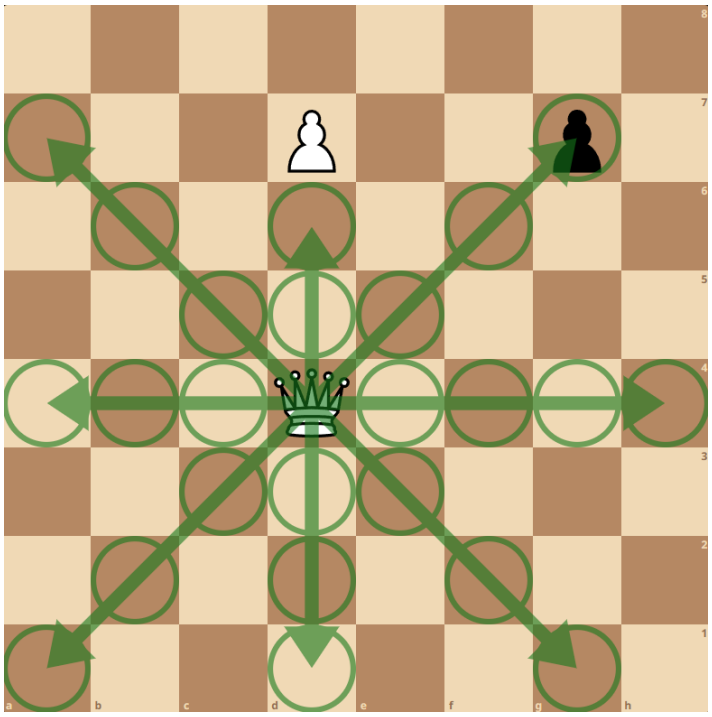

Figura 3.2: Movimiento de la reina

Para generar sus movimientos vamos a seguir un procedimiento similiar al que hemos planteado para el caballo. En este caso vamos a precalcular si nos encontraramos en una casilla determinada. Por ejemplo, la casilla en la que se encuentra la reina de la Figura 3.2 es la casilla *D4*, si ignoramos ambos peones entonces el número de casillas

que hay hacia arriba es 4, a izquierda 3, abajo 3, derecha 4, en la diagonal arriba-derecha 4, en la diagonal arriba-izquierda 3, en la diagonal abajo-izquierda 3, y en la diagonal abajo-derecha 3.

A continuación podemos ver el código desarrollado para dichos movimientos:

```
int[][] preCalculatedSquaresToEdge = new int[64][];

for (int j = 0; j < 8; j++)
{
    for (int i = 0; i < 8; i++)
    {
        int index = i + j * 8;

        // squares to edge

        int up = j;
        int down = 7 - j;
        int left = i;
        int right = 7 - i;

        int d1 = Math.Min(up, right);
        int d2 = Math.Min(up, left);
        int d3 = Math.Min(down, left);
        int d4 = Math.Min(down, right);

        preCalculatedSquaresToEdge[i + j * 8] = new int[8] {
            up, left, down, right, d1, d2, d3, d4
        };
    }
}
```

Fragmento de código 3.3: Precalculación del numero de cuadrados en las 8 direcciones

Una vez tenemos la *LookUpTable preCalculatedSquaresToEdge*, dependiendo de si la pieza es una torre, un alfil o una reina, comprobaremos las casillas en las direcciones correspondientes, para ver si se encuentran vacias, o si contienen alguna pieza. Si contienen una pieza del mismo color descartaremos esa casilla como válida y no seguiremos comprobando mas casillas en esa dirección. En cambio, si contiene una pieza del color contrario, sí que lo añadiremos como movimiento válido, puesto que sería una captura, pero también dejaremos de comprobar las siguientes casillas en esa dirección, como en la Figura 3.2.

Podemos ver aquí finalmente el código que planteamos para el movimiento de la torre/alfil/reina.

```
private static List<Move> GenerateSlidingMoves(Board board, int index)
{
    List<Move> moves = new List<Move>();

    //... comprobar si la pieza es una torre/alfil/reina
    //... dependiendo de la pieza ver que direcciones hay que comprobar

```

```
8       for (//... las direcciones pertinentes ...//)
9       {
10          int n = preCalculatedSquaresToEdge[index][direccion];
11
12          for (int i = 0; i < n; i++)
13          {
14              int targetIndex = index + directionOffsets[d] * (i + 1);
15
16              Piece targetPiece = board.GetPiece(targetIndex);
17
18              if (targetPiece.type == Piece.Type.None)
19              {
20                  // ... anadimos el movimiento
21              }
22              else
23              {
24                  if (targetPiece.color != piece.color)
25                  {
26                      // ... anadimos el movimiento
27                  }
28
29                  break;
30              }
31          }
32      }
33
34      return moves;
35  }
```

Fragmento de código 3.4: Generación del movimiento de la torre/alfil/reina (simplificado)

3.1.3. El Peón

Los peones se pueden desplazar únicamente hacia adelante, siempre que esa casilla se encuentre vacía. Adicionalmente, si el peón se encuentra en la fila 2, para el caso de los peones blancos, se puede desplazar 2 casillas hacia adelante. Podemos ver un ejemplo en la Figura 3.3. Hay que resaltar también que si un peón llega a la última fila se puede promocionar a una de las siguientes piezas: reina, torre, álfil y caballo.

Para estos movimientos no necesitaremos precalcular nada, puesto que sólo necesitamos comprobar la casilla de adelante. Simplemente tendremos que comprobar si se encuentra vacía y si el peón se encuentra en fila 2. En ese caso tendremos que comprobar también si el peón se puede desplazar 2 casillas hacia adelante, siempre y cuando no haya ninguna casilla ocupada a lo largo de ese desplazamiento. Aquí podemos ver la parte del código que genera estos movimientos, únicamente para el peón blanco, el peón negro sigue las mismas reglas sólo que en la dirección contraria:

```
1 // double push
2
```

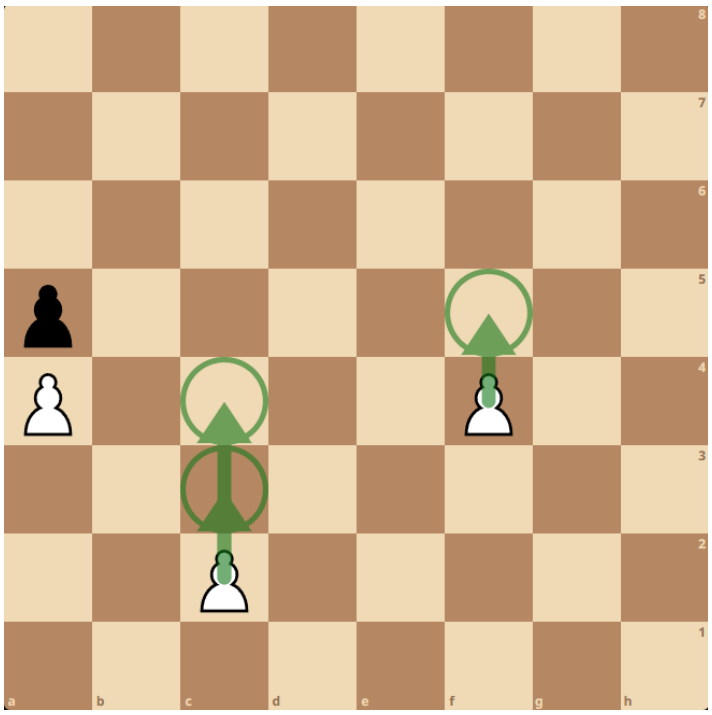

Figura 3.3: Movimiento del peón

```
3  if (j == 6)
4  {
5      for (int jj = 0; jj < 2; jj++)
6      {
7          int targetIndex = index + (jj + 1) * directionOffsets[Direction.Up];
8          Piece targetPiece = board.GetPiece(targetIndex);
9
10         if (targetPiece.type == Piece.Type.None)
11         {
12             moves.Add(new Move
13             {
14                 ...
15                 flags = (jj == 0) ? Move.Flags.None : Move.Flags.DoublePush
16             });
17         }
18         else
19         {
20             // if there is a piece in between then you cant double push
21
22             break;
23         }
24     }
25 }
26 else if (j < 6 && j > 0) // single push
27 {
28     int targetIndex = index + directionOffsets[Direction.Up];
29     Piece targetPiece = board.GetPiece(targetIndex);
30
31     if (targetPiece.type == Piece.Type.None)
32     {
33         moves.Add(new Move
34         {
```

```
35              ...
36              flags = (j == 1) ? Move.Flags.Promotion : Move.Flags.None,
37              promotionPieceType = board.PromotionPieceType
38          });
39      }
40  }
```

Fragmento de código 3.5: Parte del código para la generación del movimiento del peón

Tenemos que realizar una comprobación para conocer si el peón se encuentra en la fila 2, en este caso se comprueba que la coordenada j de la pieza sea igual a 6, esto se debe a que de la manera en la que hemos construido el array del tablero las filas irían numeradas desde la fila 0 en la parte superior hasta la fila 7 en la parte inferior. Si es ese el caso, entonces tendremos que comprobar si se puede desplazar hacia adelante 1 ó 2 casillas. En el caso en el que se pueda desplazar 2 casillas, es decir, no hay ninguna pieza que obstaculice el camino, tendremos que marcar ese movimiento con el *flag* *DoublePush* para indicar que ese movimiento realiza un movimiento doble de un peón. En cambio, si el peón se encuentra en una fila diferente sólo tendremos que comprobar si se puede mover una casilla hacia adelante, con la excepción de que si se desplaza a la última fila, entonces tendremos que marcar el *flag* *Promotion* para indicar que ese movimiento realiza una promoción de un peón.

Los movimientos de captura son diferentes de sus movimientos normales. Mientras que los peones se mueven hacia adelante, en línea recta, capturan las piezas en las diagonales hacia adelante. Adicionalmente, si un peón enemigo se mueve 2 casillas en un turno, al siguiente turno se puede capturar como si hubiese movido una sola casilla. Podemos ver un ejemplo en la Figura 3.4.

Para el caso de las capturas sí que hemos precalculado los movimientos, puesto que así podemos ahorrarnos tiempo en ciertas comprobaciones. En concreto, para comprobar si el peón se encuentra en los bordes del tablero y a la hora de comprobar si puede capturar arriba a la izquierda o a la derecha. Simplemente con la *LookUpTable* de las capturas precalculadas de la casilla en la que se encuentra nuestro peón, tendremos que comprobar que exista una pieza del color opuesto en las casillas disponibles para que se pueda efectuar ese movimiento.

Finalmente para la **captura al paso**, tendremos que comprobar que justo en el anterior movimiento que se realizó en el tablero se hizo un movimiento doble de peón por parte del rival. Aquí entra todo lo que explicamos previamente acerca del sistema del estado del tablero en la Sección 2.3.

Aquí podemos ver el código que genera las capturas al paso:

```
1 ref readonly BoardState boardState = ref board.GetBoardState();
2
```

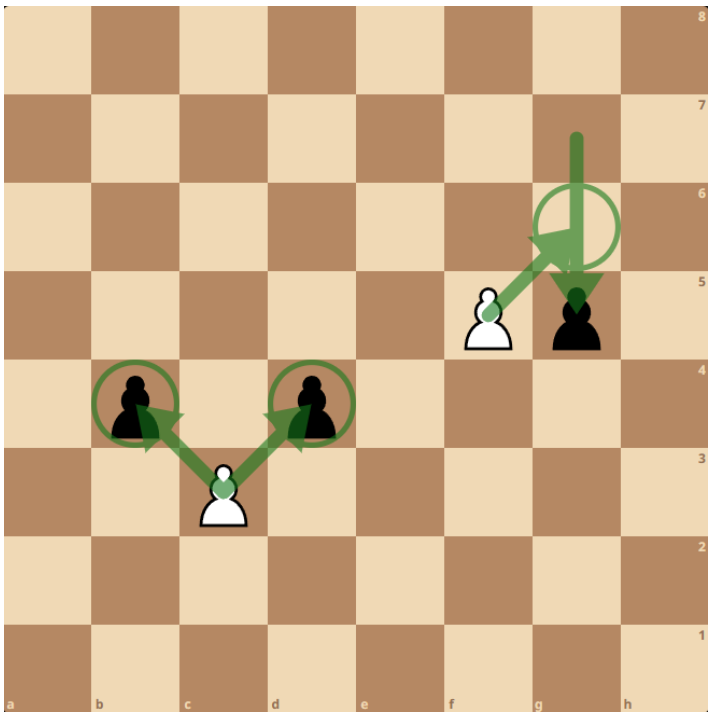

Figura 3.4: Movimientos de captura del peón

```
if (boardState.IsEnPassantAvailable())
{
    int enPassantSquareI = boardState.GetEnPassantSquareIndex() % 8;

    if (j == 3)
    {
        if (i + 1 == enPassantSquareI)
        {
            int targetIndex = index + directionOffsets[(int)Direction.D1];
            moves.Add(new Move
            {
                ...
                pieceTarget = Piece.NullPiece,
                flags = Move.Flags.EnPassant
            });
        }
        else if (i - 1 == enPassantSquareI)
        {
            int targetIndex = index + directionOffsets[(int)Direction.D2];
            moves.Add(new Move
            {
                ...
                pieceTarget = Piece.NullPiece,
                flags = Move.Flags.EnPassant
            });
        }
    }
}
```

Fragmento de código 3.6: Parte del código de la generación de los movimientos de captura al paso del peón

Realizamos una consulta al estado actual del tablero, y comprobamos si es posible realizar una captura al paso. En ese caso comprobamos si nuestro peón se encuentra en la fila 5, que es la única fila en la que se pueden realizar este tipo de capturas, recordemos que ésto es sólo para el caso de los peones blancos, se realizaría de una forma similar para los peones negros, solo que en direcciónes opuestas. Si se da ese caso entonces comprobamos si la casilla de la diagonal superior izquierda o derecha concuerdan con la casilla correspondiente a la captura al paso y si se verifica entonces la captura al paso es posible, se contruye el movimiento con el *flag EnPassant* y se añade a la lista de movimientos posibles.

Finalmente todos estos fragmentos que hemos visto para generar estos posibles movimientos posibles del peón, se recogen en una función, al igual que con el resto de piezas.

```
1 private static List<Move> GeneratePawnMoves(Board board, int index);
```

Fragmento de código 3.7: Función que genera los movimientos del peón

3.1.4. El Rey

El rey se puede mover en cualquiera de las 8 direcciones, pero sólo una casilla de distancia. Para generar estos movimientos hemos seguido el mismo procedimiento utilizado con otras piezas, es decir, precalcular las casillas disponibles a las que se puede mover el rey por cada casilla del tablero y guardarlas en una *LookUpTable* para consultarla a la hora de generar los movimientos.

Adicionalmente el rey es capaz de realizar un movimiento especial denominado **enroque**, el cual consiste en mover simultáneamente al rey y a una de las torres, de manera que el rey se desplace dos casillas hacia la torre y ésta se coloque en la casilla inmediatamente al otro lado del rey. Este movimiento tiene ciertas condiciones: no debe haber piezas entre el rey y la torre, ni el rey ni la torre deben haber sido movidos previamente, el rey no puede estar en jaque, ni puede atravesar ni terminar en una casilla amenazada por una pieza enemiga.

Para implementar el enroque necesitaremos conocer qué casillas está controlando o *atacando* el color contrario, puesto que las reglas nos lo exigen al tener que comprobar que el rey no se encuentra en jaque y a la hora de enrocarse no puede atravesar casillas que están siendo *atacadas*. Para ello implementaremos una función, que para un color de piezas determinado, nos devolverá un *bitmap* del tablero indicando con un 1 si la casilla está siendo atacada por alguna pieza del color que hayamos seleccionado, ó 0 en el caso contrario. A continuación mostramos la implementación de esta función.

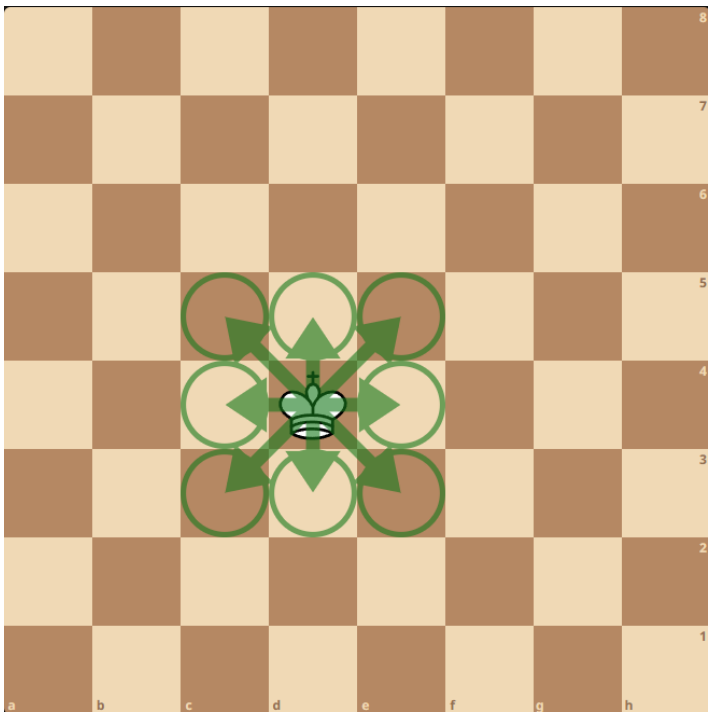

Figura 3.5: Movimiento del rey

```
ulong GetControlledSquaresBitboardByColor(Board board, Piece.Color color)
{
    ulong squares = 0;

    List<int> piecesIndices = board.GetPiecesIndices(color);

    foreach (int index in piecesIndices)
    {
        Piece piece = board.GetPiece(index);

        switch (piece.type)
        {
            case Piece.Type.Pawn:
                // ... SOLO MOVIMIENTOS DE CAPTURAS DIAGONALES
            case Piece.Type.Knight:
                // ... IGUAL QUE ANTES
            case Piece.Type.Bishop:
            case Piece.Type.Rook:
            case Piece.Type.Queen:
                // ... IGUAL QUE ANTES
            case Piece.Type.King:
                // ... SOLO MOVIMIENTOS "NORMALES" SIN INCLUIR ENROQUE
        }
    }

    return squares;
}
```

Fragmento de código 3.8: Función que devuelve un bitmap del tablero con las casillas controladas por las piezas del color seleccionado

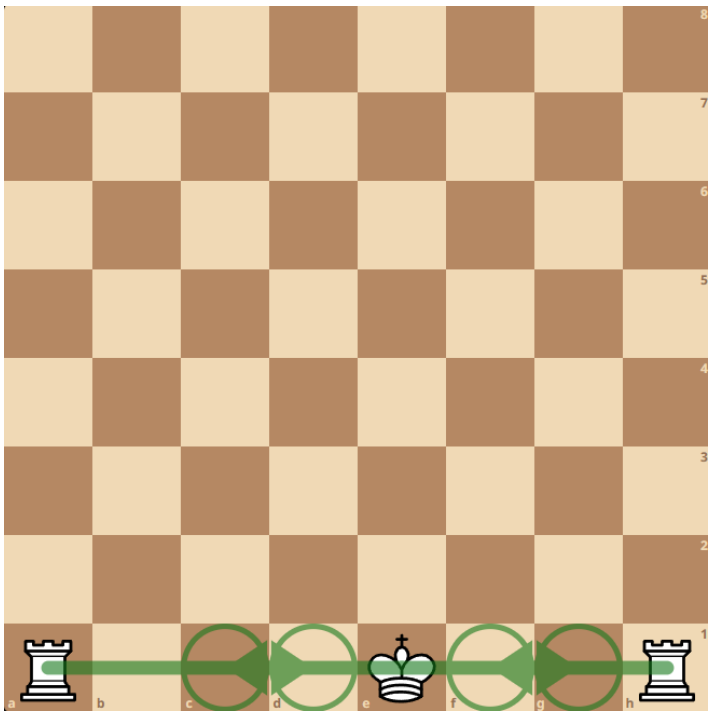

Figura 3.6: Movimiento de enroque del rey

Simplemente tomamos todas las piezas del color escogido y calculamos para cada una de ellas los índices de las casillas que están atacando, similarmente a cuando generabamos los movimientos, sólo que esta vez únicamente nos importan las posiciones de las casillas. Podríamos marcar estos índices en un array de *booleanos* de dimensión 64 y todo funcionaría perfectamente. Sin embargo, podemos utilizar un *unsigned long*, que es un entero sin signo de 64 *bits* y que es mucho mas eficiente que usar un array, al tener 64 *bits* nos encaja perfectamente con el número de casillas del tablero, así que podemos ir modificando los *bits* de este entero haciendo que cada *bit* sea una casilla y marcándolo como 1 ó 0 en función de si esta siendo atacada o no. Para modificar los *bits* de este entero utilizaremos lógicamente las operaciones a nivel de *bit*: *and*, *or* y desplazamientos de *bits*. Podemos ver un ejemplo en la Figura 3.7.

Todas las casillas marcadas por un círculo verde son aquellas casillas que están siendo controladas por las piezas negras, en este caso, la función *GetControlledSquaresBitboardByColor* (con el color negro) nos devolveria el siguiente entero de 64 *bits*:

```
1011010001111100110111110111000010101000000001000101001000000001
```

Empezando por los 8 *bits* mas significativos *10110100* vemos que cuadra con las casillas controladas de la primera fila (fila 8 de la Figura 3.7) y lo mismo para los siguientes grupos de 8 *bits* correspondientes al resto de filas. Si queremos mas tarde

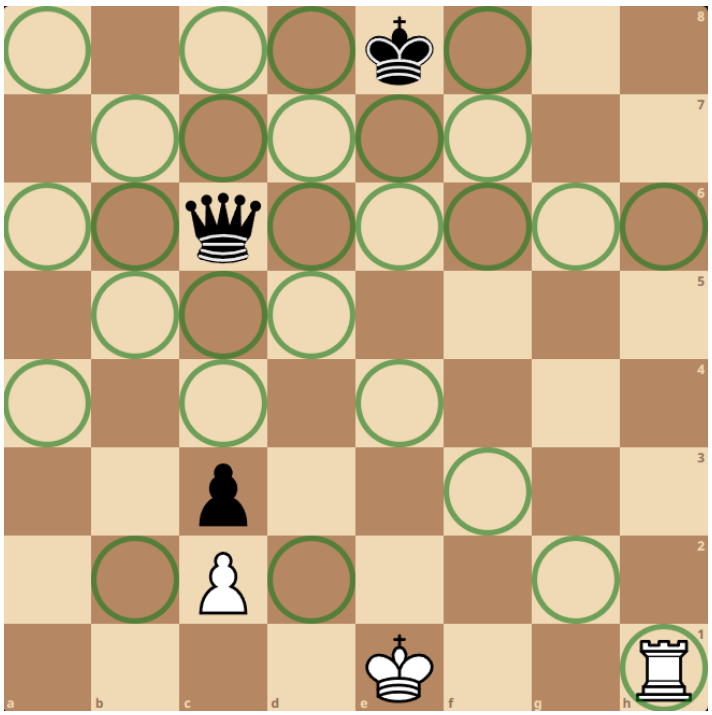

Figura 3.7: Bitmap con las casillas atacadas por las piezas negras

comprobar si la casilla *n-esima* está siendo atacada, simplemente comprobamos que si realizamos una operación *and* con un *bit* desplazado n veces hacia la izquierda da distinto de 0:

```
(bitmap & (1UL <<n)) != 0
```

Una vez ya sabemos cómo obtener las casillas atacadas por un color determinado de pieza, continuamos con la implementación del enroque:

```
1  ref readonly BoardState boardState = ref board.GetBoardState();
2  bool canCastleShort = boardState.CanCastleShort(piece.color);
3  bool canCastleLong = boardState.CanCastleLong(piece.color);
4
5  if (canCastleShort || canCastleLong)
6  {
7      ulong bitmap = GetControlledSquaresBitboardByColor(
8          board,
9          Piece.GetOppositeColor(piece.color)
10     );
11
12     // first check if the king is in check
13
14     bool isKingInCheck = (bitmap & (1UL << index)) != 0;
15
16     // if the king is not in check then
17
18     if (!isKingInCheck)
19     {
20         if (canCastleShort)
21         {
```

```
22              // check squares in between
23
24              bool isShortCastleLegal = true;
25
26              foreach (int squareIndex in shortCastleIndices[piece.color - 1])
27              {
28                  Piece targetPiece = board.GetPiece(squareIndex);
29
30                  if (
31                      ((bitmap & (1UL << squareIndex)) != 0) ||
32                      targetPiece.type != Piece.Type.None
33                  )
34                  {
35                      isShortCastleLegal = false;
36                      break;
37                  }
38              }
39
40              if (isShortCastleLegal)
41              {
42                  moves.Add(new Move
43                  {
44                      squareSourceIndex = index,
45                      squareTargetIndex = shortCastleTargetKingIndex[piece.color -
46                      pieceSource = piece,
47                      pieceTarget = Piece.NullPiece,
48                      flags = Move.Flags.CastleShort
49                  });
50              }
51          }
52
53          //... lo mismo para el enroque largo
54      }
55  }
```

Fragmento de código 3.9: Codigo de generación del enroque corto

Empezamos obteniendo el estado actual del tablero para comprobar si está disponible el enroque, estos *flags* tanto para el enroque corto como el enroque largo nos indican si el enroque está disponible, es decir, si el rey o la torre del correspondiente lado no se han movido de acuerdo con las reglas. Pero no si se puede realizar en este momento debido a que podrían existir piezas en el camino entre el rey o la torre o esas mismas casillas podrían estar siendo atacadas, incluida la del rey. Estos son los diferentes casos adicionales que tendremos que comprobar.

Para el caso en el que sí esté disponible el enroque, obtenemos las casillas controladas por las piezas del color contrario utilizando la función *GetControlledSquaresBitboardByColor* que hemos implementado y comprobamos si el rey se encuentra en una de las casillas atacadas. Si no se encuentra en una de ellas entonces comprobamos si se puede realizar cada uno de los enroques, corto y largo, comprobando que esten libres las casillas entre el rey y la torre y que no esten siendo

atacadas. Si todo esto se cumple entonces añadimos el movimiento e indicamos el *flag* del movimiento como *CastleShort* o *CastleLong* dependiendo del enroque.

3.1.5. De movimientos pseudo-legales a movimientos legales

Hasta ahora los movimientos que hemos ido generando no son movimientos legales sino que son pseudo-legales, puesto que en ningún momento hemos comprobado si el rey se encuentra en jaque, después de realizar el movimiento. Veamos un ejemplo de ésto en la Figura 3.8

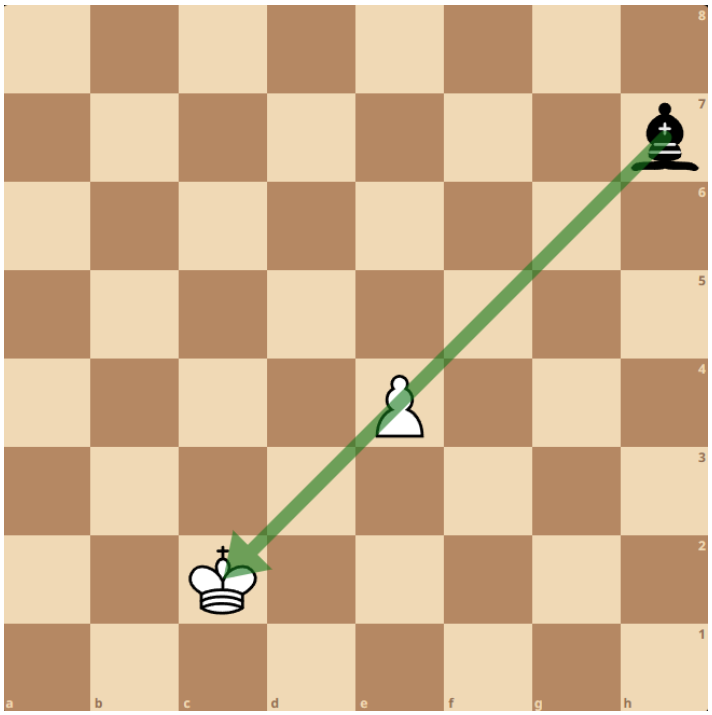

Figura 3.8: Ejemplo clavada de peón

En este escenario, las piezas blancas no pueden mover el peón, puesto que ello revelaria un ataque directo del álfil negro al rey, se dice entonces que el peón está **clavado**, puesto que no se puede mover. Con nuestro codigo actual mover el peón sería considerado erróneamente un movimiento "válido", por lo que tenemos que tener en cuenta este tipo de situaciones.

Para solucionar este problema lo que haremos será generar los movimientos de la pieza, y uno por uno jugarlos en el tablero y comprobar entonces si el rey se encuentra en una casilla atacada. Si no se encuentra atacado entonces cogemos ese movimiento y lo añadimos a la lista de movimientos legales, finalmente deshacemos el movimiento del tablero.

Aquí podemos ver el código:

```
1 public static List<Move> GetLegalMoves(Board board, int index)
```

```
{
    List<Move> legalMoves = new List<Move>();
    List<Move> pseudoLegalMoves = GetPseudoLegalMoves(board, index);

    Piece piece = board.GetPiece(index);

    foreach (Move move in pseudoLegalMoves)
    {
        // make the move

        board.MakeMove(move, true);

        // check if after the move the king is in check

        if (!IsKingInCheck(board, piece.color))
        {
            legalMoves.Add(move);
        }

        // undo the move

        board.UndoMove(true);
    }

    return legalMoves;
}
```

Fragmento de código 3.10: Generación de los movimientos legales de una pieza

De esta forma podemos filtrar aquellos movimientos que en un principio parecerían válidos, pero que en realidad no lo son. La función *GetPseudoLegalMoves* retorna los movimientos pseudo-legales de una pieza en función de su tipo. Esta función simplemente invoca, según el tipo de pieza, a la función correspondiente que genera los movimientos de dicho tipo, tal como hemos visto a lo largo de este capítulo.

Aquí podemos ver el código de dicha función:

```
public static List<Move> GetPseudoLegalMoves(Board board, int index)
{
    Piece piece = board.GetPiece(index);

    switch (piece.type)
    {
        case Piece.Type.Pawn:
            return GeneratePawnMoves(board, index);
        case Piece.Type.Knight:
            return GenerateKnightMoves(board, index);
        case Piece.Type.Bishop:
        case Piece.Type.Queen:
        case Piece.Type.Rook:
            return GenerateSlidingMoves(board, index);
        case Piece.Type.King:
            return GenerateKingMoves(board, index);
    }

    return new List<Move>();
}
```

Fragmento de código 3.11: Función GetPseudoLegalMoves

3.1.6. Encapsulación

Todas las funciones que hemos ido desarrollando con las *LookUpTables*, que se han ido precalculando, las hemos agrupado en una clase estática de C#. Con esta **decisión de diseño** avalamos el hecho de que estas funciones no requieren de un estado específico de la instancia de la clase [14], a la que hemos llamado *MoveGeneration*, esta clase simplemente actuará de contenedor para las funciones y tablas que hemos mencionado:

```
1  public static class MoveGeneration
2  {
3      // LookUpTables
4      private static int[][] preCalculatedKnightMoves = new int[64][];
5      // ...
6
7      static MoveGeneration()
8      {
9          // ... pre-calculo de las LookUpTables en el constructor
10     }
11
12     // las funciones que generan los movimientos de las piezas
13     private static List<Move> GenerateKnightMoves(Board board, int index)
14     {
15         // ...
16     }
17     // ...
18
19     // funcion que genera los movimientos pseudo-legales de una pieza
20     public static List<Move> GetPseudoLegalMoves(Board board, int index)
21     {
22         // ...
23     }
24
25     // funcion que genera los movimientos legales de una pieza
26     public static List<Move> GetLegalMoves(Board board, int index)
27     {
28         // ...
29     }
30 }
```

Fragmento de código 3.12: Clase MoveGeneration

Esta encapsulación que hemos realizado nos permite de manera muy sencilla obtener una lista de los movimientos legales de una pieza con solo pasarle el tablero, y el índice de la casilla (0 a 63) donde se encuentra la pieza, de la siguiente forma:

```
1  List<Move> moves = MoveGeneration.GetLegalMoves(board, index);
```

3.2. Jaque mate y Rey ahogado

Una vez que podemos determinar los movimientos de cada pieza, resulta relativamente fácil verificar las condiciones que indican que una partida ha concluido debido a 2 posibles escenarios: jaque mate y rey ahogado o tablas.

El **jaque mate** se da cuando el rey del oponente está bajo amenaza directa de captura (en jaque) y no tiene ninguna forma de escapar, ya sea moviéndose a una casilla segura, bloqueando la amenaza con otra pieza, o capturando la pieza que amenaza al rey. En este punto, el juego termina y el jugador que ha puesto en jaque mate al rey adversario es declarado ganador [1].

El **rey ahogado** es similar al jaque mate, sólo que en este caso no está en jaque, pero no puede realizar ningún movimiento legal porque todas sus casillas de escape están bloqueadas por sus propias piezas o controladas por las piezas del oponente. Esto resulta en un empate (tablas) en lugar de una victoria para el oponente [1].

Para realizar ambas comprobaciones, lo que haremos será generar todos los movimientos legales de todas las piezas del color al que le toca mover. Si el numero total de movimientos generados es 0, entonces obtendremos las casillas que está atacando el rival y comprobaremos si la casilla en la que se encuentra el rey está siendo atacada o no. Si se encuentra atacada, entonces se declarara que el jugador que hizo el movimiento previo gana por jaque mate. En cambio, si no se encuentra atacado el rey se declarara final por rey ahogado, es decir, por tablas.

Para ello, si recordamos en la Sección 2.5, donde hablabamos sobre la clase Game, la máquina de estados, y las transiciones entre estos (apartado 2.5.2), podemos añadir estas comprobaciones en el estado *NextTurn*, para que justo antes de pasarle el turno al siguiente jugador se compruebe si ha habido jaque mate o rey ahogado.

Aquí podemos ver el código modificado del estado *NextTurn*, el cual implementa las comprobaciones mencionadas antes:

```
1  Piece.Color turnColor = board.GetTurnColor();
2
3  bool isGameOver = false;
4
5  // COMPROBACION DE JAQUE MATE/REY AHOGADO
6  List<Move> moves = MoveGeneration.GetAllLegalMovesByColor(board, turnColor);
7
8  if (moves.Count == 0)
9  {
10     if (MoveGeneration.IsKingInCheck(board, turnColor))
11     {
12         switch (turnColor)
13         {
14             case Piece.Color.White:
15                 // JAQUE MATE DE LAS PIEZAS NEGRAS
```

```
16                  break;
17              case Piece.Color.Black:
18                  // JAQUE MATE DE LAS PIEZAS BLANCAS
19                  break;
20          }
21      }
22      else
23      {
24          // REY AHOGADO
25      }
26
27      isGameOver = true;
28  }
29
30  if (!isGameOver)
31  {
32      // next player turn
33
34      playerToMove = turnColor == Piece.Color.White ? playerWhite : playerBlack;
35      playerToMove.NotifyTurnToMove();
36      gameState = GameState.PlayerTurn;
37  }
38  else
39  {
40      gameState = GameState.Over;
41  }
```

Fragmento de código 3.13: Estado NextTurn con la comprobación de jaque mate y rey ahogado

Por el momento, independientemente del motivo por el cual se termine el juego, vease jaque mate o rey ahogado simplemente transicionamos al estado *Over*. Más adelante, cuando implementemos la interfaz de usuario señalizaremos con un texto el motivo por el cual ha terminado el juego.

3.3. Tablas por triple repetición

En ajedrez, las tablas por triple repetición se producen cuando la misma posición exacta de las piezas en el tablero aparece tres veces durante el transcurso de una partida. Para que una posición sea considerada idéntica, se deben cumplir ciertas condiciones:

1. Las mismas piezas del mismo color ocupan las mismas casillas.

2. Las mismas posibles jugadas están disponibles para cada jugador (es decir, los derechos de enroque y captura al paso no deben haber cambiado entre las repeticiones).

Cuando se repite la misma posición por tercera vez, cualquiera de los jugadores puede reclamar tablas. Esta regla está diseñada para evitar partidas interminables y

es una de las formas en que una partida puede terminar en empate. Por simplificación consideraremos tablas automaticamente una vez se repitan tres posiciones.

Para implementar esta regla necesitamos obtener y comparar estas posiciones incluyendo el estado del tablero. Para ello existen diferentes técnicas que se explican a continuación.

3.3.1. Notación de Forsyth-Edwards

La notación Forsyth Edwards[1] (FEN) es un estándar para describir una posición específica en una partida de ajedrez. Esta notación es esencial para registrar y comunicar posiciones de ajedrez de manera precisa y concisa. Una cadena FEN la expresaremos con cuatro campos separados por espacios: la posición de las piezas, el turno de juego, los derechos de enroque y la posibilidad de captura al paso. Aunque existen más parámetros, los omitiremos por simplicidad puesto que no los hemos considerado relevantes ya que están ligados a reglas que no hemos implementado.

1. **Posición de las piezas**:
   - Este campo describe la ubicación de todas las piezas en el tablero, fila por fila desde la octava fila (fila superior) hasta la primera fila (fila inferior).
   - Las piezas se representan con letras: `p` (peón), `r` (torre), `n` (caballo), `b` (alfil), `q` (dama), `k` (rey). Las letras mayúsculas representan piezas blancas y las minúsculas representan piezas negras.
   - Las casillas vacías se representan con números del `1` al `8`, indicando cuántas casillas vacías hay consecutivamente.
2. **Turno de juego**:
   - Representado por una `w` si es el turno de las blancas o una `b` si es el turno de las negras.
3. **Derechos de enroque**:
   - Representado por las letras `K` (enroque corto blanco), `Q` (enroque largo blanco), `k` (enroque corto negro) y `q` (enroque largo negro). Si ninguno de los bandos puede enrocar, se usa un `-`.
4. **Posibilidad de captura al paso**:

[1] `https://en.wikipedia.org/wiki/ForsythEdwards_Notation`

– Representado por la notación de la columna en la que es posible una captura al paso. Si no hay posibilidad de captura al paso, se usa un `-`.

Veamos un ejemplo con la posición inicial del tablero:

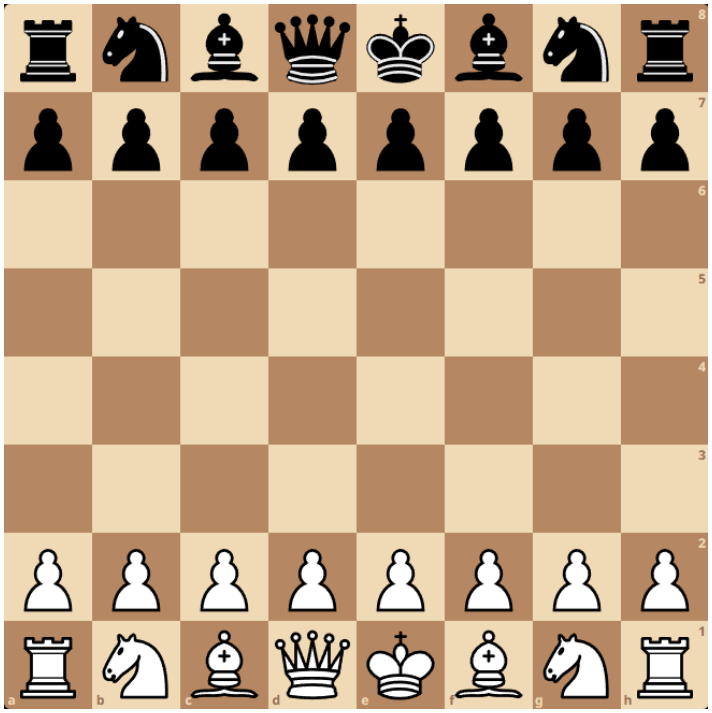

Figura 3.9: Posición inicial de un tablero de ajedrez

Cuya notación FEN correspondería a:

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -
```

Para poder construir esta cadena FEN, tendríamos que recorrer todas las filas del tablero en busca de las piezas que en él se encuentran, y consultar el estado del tablero para conocer el turno actual, si se puede realizar una captura al paso y en que casilla, y si estan disponibles los derechos de enroque.

Con la cadena FEN generada, ya podríamos realizar comparaciones entre posiciones y verificar si una misma posición se ha repetido tres veces durante el juego. No obstante, generar una cadena FEN es un proceso relativamente lento ya que requiere revisar todo el tablero en busca de las piezas. Esto no sería relevante para este caso específico, ya que generaríamos una sola cadena por cada movimiento en el tablero para su posterior comparación. Sin embargo, cuando desarrollemos la *IA*, necesitaremos producir muchas más cadenas y requeriremos que el proceso sea lo más rápido posible, por lo que este método resultaría inviable.

3.3.2. Zobrist Hashing

Zobrist hashing es una técnica utilizada en programación de juegos de ajedrez (y otros juegos de tablero) para representar de manera eficiente y rápida la posición en el tablero. Fue inventada por Albert Zobrist en 1970 y se utiliza principalmente para crear claves únicas (*hashes*) que representan las posiciones del tablero [15].

En este método, transformamos una posición del tablero en una clave *hash*, en nuestro caso un número de 64 bits (*unsigned long*). Esto genera un pequeño inconveniente: un número de 64 bits tiene 2^{64} combinaciones posibles, lo que significa que podemos representar hasta 2^{64} posiciones diferentes de un tablero de ajedrez. No obstante, las combinaciones posibles de un tablero de ajedrez son mucho mayores, lo que significa que es posible que diferentes posiciones del tablero generen el mismo *hash*. Con un número de 64 bits podemos esperar una colisión cada 2^{32} o 4 billones de posiciones [15]. Aunque no podemos evitar este problema de colisiones, en la práctica no suele ser un inconveniente significativo.

Su funcionamiento es el siguiente:

1. **Generación de Números Aleatorios**:
   - Se crea una tabla de números aleatorios. Cada pieza en cada posición del tablero tiene un número aleatorio asociado.
   - Para un tablero de ajedrez estándar, se necesitan números aleatorios para 12 tipos de piezas (6 para blancas y 6 para negras) y 64 casillas del tablero. Esto da un total de 768 números aleatorios (12 piezas $\times$ 64 casillas).

   Aquí podemos ver el código que pre-genera estos números aleatorios para cada una de las 64 casillas y a su vez por cada una de las 12 piezas, los cuales se almacenan en una tabla a la que hemos llamado *pieceKeys*:

```
ulong[,] pieceKeys = new ulong[64, 12];

for (int i = 0; i < 64; i++)
{
    for (int j = 0; j < 12; j++)
    {
        pieceKeys[i, j] = (ulong)random.NextInt64();
    }
}
```

Fragmento de código 3.14: Números aleatorios para las piezas

   También se generan numeros aleatorios asociados al color del turno, a la captura al paso y a los derechos de enroque por cada color siguiendo el mismo procedimiento.

2. **Cálculo del Hash Inicial**:

   – Se inicializa el valor del *hash* en cero.

   – Para cada pieza en el tablero, se toma el número aleatorio correspondiente a esa pieza en su posición actual y se realiza una operación **XOR** con el valor del *hash*.

   – La operación **XOR** asegura que cada pieza y posición contribuyan de manera única al valor del *hash*, y que la adición y eliminación de piezas sean operaciones reversibles.

   – Tambien se realiza la operacion **XOR** con los números aletorios correspondientes al color del turno actual, a los derechos de enroque que son posibles por ambos colores y en caso de que esté disponible el número correspondiente a la captura al paso.

Aquí podemos ver el código de la función *GetKey* con la que podemos obtener el hash de una posición del tablero utilizando las tablas de numeros aleatorios precalculadas anteriormente:

```
public static ulong GetKey(Board board)
{
    ulong hashKey = 0;

    // piezas blancas
    foreach (int i in board.GetPiecesIndices(Piece.Color.White))
    {
        Piece piece = board.GetPiece(i);

        hashKey ^= pieceKeys[i, (int)(piece.type - 1)];
    }

    // piezas negras ...

    // board state
    ref readonly BoardState boardState = ref board.GetBoardState();

    // captura al paso
    if (boardState.IsEnPassantAvailable())
    {
        hashKey ^= enPassantKey;
    }

    // color del turno

    hashKey ^= turnColorKeys[(int)boardState.GetTurnColor() - 1];

    // enroque ...

    return hashKey;
}
```

Fragmento de código 3.15: Función GetKey

Con esta función, calcularemos el *hash* del tablero únicamente la primera vez que se inicie el juego. Posteriormente, actualizaremos el *hash* conforme se realicen movimientos en el tablero, aprovechando las propiedades de la operación **XOR**.

3. **Actualización del Hash**:
   - Cuando una pieza se mueve, el *hash* se actualiza realizando una operación **XOR** para eliminar la pieza de su posición inicial y otra operación **XOR** para colocarla en su nueva posición. Igualmente sucede cuando cambian los derechos de enroque, el color del turno o la captura al paso.
   - Esto permite actualizar el *hash* sin tener que recalcularlo entero, lo que es más eficiente y rápido.

A la clase *Board* que vimos en la sección 2.3 le añadiremos un atributo que llamaremos *zobrist* el cual corresponderá al *hash* del tablero y el cual irá cambiando a la vez que se realizan movimientos. Veamos las modificaciones que hemos realizado a la hora de colocar piezas en el tablero:

```
1  private void SetPiece(int index, Piece piece)
2  {
3      // obtenemos la pieza que habia antes en el tablero
4      Piece piecePrevious = pieces[index];
5
6      if (piecePrevious.type != Piece.Type.None)
7      {
8          // "quitamos" la pieza que habia antes del hash
9          zobrist ^= ZobristHashing.GetPieceKey(index, piecePrevious);
10     }
11
12     // la nueva pieza
13     if (piece.type != Piece.Type.None)
14     {
15         // "colocamos" la nueva pieza en el hash
16         zobrist ^= ZobristHashing.GetPieceKey(index, piece);
17     }
18
19     // ponemos la pieza en el tablero
20     pieces[index] = piece;
21 }
```

Fragmento de código 3.16: Función SetPiece modificada

De esta manera vamos actualizando el *hash* a la hora de mover las piezas. La función *GetPieceKey* de la clase estática *ZobristHashing* simplemente devuelve el número aleatorio correspondiente a la pieza seleccionada para la casilla

seleccionada. De manera similar, debe actualizarse cuando cambia el turno, así como los derechos de enroque y la captura al paso.

Finalmente, para llevar a cabo el seguimiento de todas las posiciones que se van produciendo en la partida y poder comprobar si se produce una triple repetición, utilizaremos un diccionario o *HashMap*. Este diccionario al que hemos llamado *zobristHistory*, almacena las posiciones del tablero utilizando una clave *hash* de 64 bits (*ulong*) generada mediante Zobrist *hashing*. El valor asociado a cada clave es un entero (*int*) que representa el número de veces que esa posición ha aparecido en la partida.

```
1 Dictionary<ulong, int> zobristHistory = new Dictionary<ulong, int>();
```

Fragmento de código 3.17: Diccionario *zobristHistory*

Entonces cada vez que realizamos un movimiento tenemos que buscar si ese *hash* ya existe en el diccionario. Si ya existe incrementamos el valor asociado a esa clave en uno. Esto indica que la posición ha aparecido nuevamente. Si no creamos la entrada en el diccionario y colocamos el número de veces que ha aparecido la posición a 1.

```
1  // ... se realiza el movimiento y se actualiza el hash
2
3  // modificamos el diccionario
4  if (zobristHistory.ContainsKey(zobrist))
5  {
6      zobristHistory[zobrist]++;
7  }
8  else
9  {
10     zobristHistory[zobrist] = 1;
11 }
```

Fragmento de código 3.18: Modificación del diccionario *zobristHistory*

Para terminar, antes de pasar el turno al siguiente jugador tenemos que comprobar entonces si la posicion actual se ha repetido 3 veces. Para ello añadimos la siguiente comprobación el estado *NextTurn* de la clase *Game* (sección 2.5) despues de la comprobación de jaque mate y rey ahogado.

```
1 else if (board.GetRepetitions() >= 3)
2 {
3     isGameOver = true;
4 }
```

Fragmento de código 3.19: Comprobación de triple repetición para el estado *NextTurn*

La función *GetRepetitions* de la clase *Board* únicamente devuelve el número de veces que se ha repetido la posición actual. Si se diese el caso el juego terminaría y se transicionaría al estado *Over*.

3.4. Regla de los cincuenta movimientos

La regla de los cincuenta movimientos establece que una partida acaba en tablas si cada jugador ha hecho los últimos 50 movimientos consecutivos sin que haya habido ningún movimiento de peón ni captura de pieza.

Para ello tendremos que modificar la función *MakeMove* de la clase *Board* para que adicionalmente incluya un conteo de los movimientos en los que no se ha realizado ninguna captura o movimiento de peón, reseteandolo a 0 en caso contrario. Para ello haremos uso de la variable *halfMoveCount* que vimos cuando hablabamos acerca de la clase *BoardState*.

```
1  // ...
2
3  // update half move count
4  int halfMoveCount = currentBoardState.GetHalfMoveCount() + 1;
5
6  if (move.pieceSource.type == Piece.Type.Pawn ||
7      move.pieceTarget.type != Piece.Type.None)
8  {
9      halfMoveCount = 0;
10 }
11
12 currentBoardState.SetHalfMoveCount(halfMoveCount);
```

Fragmento de código 3.20: Actualizamos el conteo de medios movimientos en la función *MakeMove*

Para terminar, antes de pasar el turno al siguiente jugador tenemos que comprobar entonces si la se han realizado por lo menos 100 medios movimientos o lo que es equivalente 50 movimientos totales. Para ello añadimos la siguiente comprobación el estado *NextTurn* de la clase *Game* (sección 2.5) despues de la comprobación de jaque mate, rey ahogado y tablas por triple repetición.

```
1 else if (board.GetHalfMoveCount() >= 100)// fifty move rule
2 {
3     isGameOver = true;
4 }
```

Fragmento de código 3.21: Comprobación de triple repetición para el estado *NextTurn*

La función *GetHalfMoveCount* de la clase *Board* únicamente devuelve el número de medios movimientos, correspondiente al almacenado en el estado actual del tablero.

Capítulo 4

Diseño y desarrollo de la interfaz de usuario

Hasta el momento, todo lo que hemos desarrollado forma parte de la estructura y de la lógica de nuestro juego de ajedrez. No obstante, para interactuar y efectuar movimientos como "jugadores humanos", es necesario desarrollar una interfaz de usuario que nos permita realizar una representación gráfica o visual del tablero de ajedrez y sus piezas, y que además nos facilite la realización de movimientos mediante la interacción con ella.

4.1. Representación gráfica del juego

La clase *Board*, que vimos en la Sección 2.3, representa la lógica y el estado del tablero, contiene las piezas, y nos permitía realizar movimientos. Lo que haremos será crear una interfaz que tome dicho tablero y lo muestre gráficamente en la pantalla. Para ello haremos uso de los objetos o nodos *Sprite2D* [16] de Godot, los cuales sirven para mostrar una imagen o *textura* en la pantalla en una posición determinada.

Como en este TFG nuestro objetivo no es enseñar cómo utilizar Godot, puesto que es simplemente una herramienta más que hemos utilizado, no explicaremos en profundidad aspectos especificos de este motor, sino que mostraremos una perspectiva global, salvo que utilicemos alguna funcionalidad muy específica que requiera de explicación.

4.1.1. La clase *BoardGraphics*

La clase *BoardGraphics* extenderá de la clase *Node2D* de Godot, ya que su función principal es manejar la representación gráfica del tablero de ajedrez y las piezas dentro de un entorno 2D. Al utilizar la clase *Node2D* como base permite que la clase *BoardGraphics* tenga acceso directo a funcionalidades específicas del entorno 2D en Godot, tales como la posición dentro de la pantalla. Esta clase se encargará de

conectarse con el tablero y renderizar los *sprites* de este mismo y de las piezas que se encuentren en él. También se encargará de dibujar los indicadores que revelan los movimientos posibles que tiene una pieza, así como indicar cuál ha sido el último movimiento.

La estructura simplificada sería la siguiente:

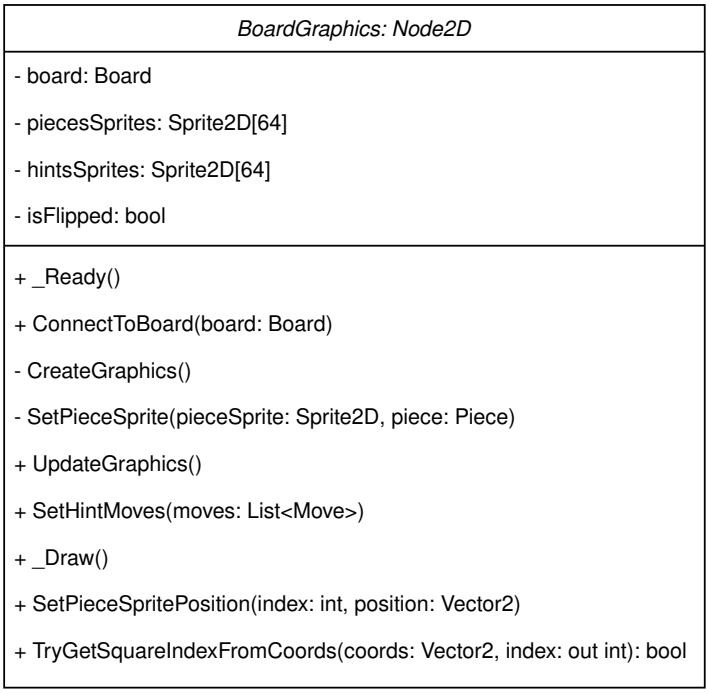

Figura 4.1: Diagrama de clases de la clase *BoardGraphics*

Veamos una explicación detallada de cada una de las partes que la componen.

El tablero

Esta clase contiene una referencia al tablero, el cual se conecta mediante la función *ConnectToBoard*. Esto nos permitirá poder acceder a toda la información relacionada con el tablero, como las piezas y cual ha sido el último movimiento realizado.

```
1 public void ConnectToBoard(Board board)
2 {
3     this.board = board;
4 }
```

Fragmento de código 4.1: Función *ConnectToBoard*

Para la representación gráfica del tablero se ha decidido utilizar un nodo *Sprite2D*, el cual se ha establecido como *hijo* del nodo *BoardGraphics*, al cual se le ha asignado la imagen de las casillas de un tablero y se ha colocado en el centro de la ventana utilizando el editor de Godot.

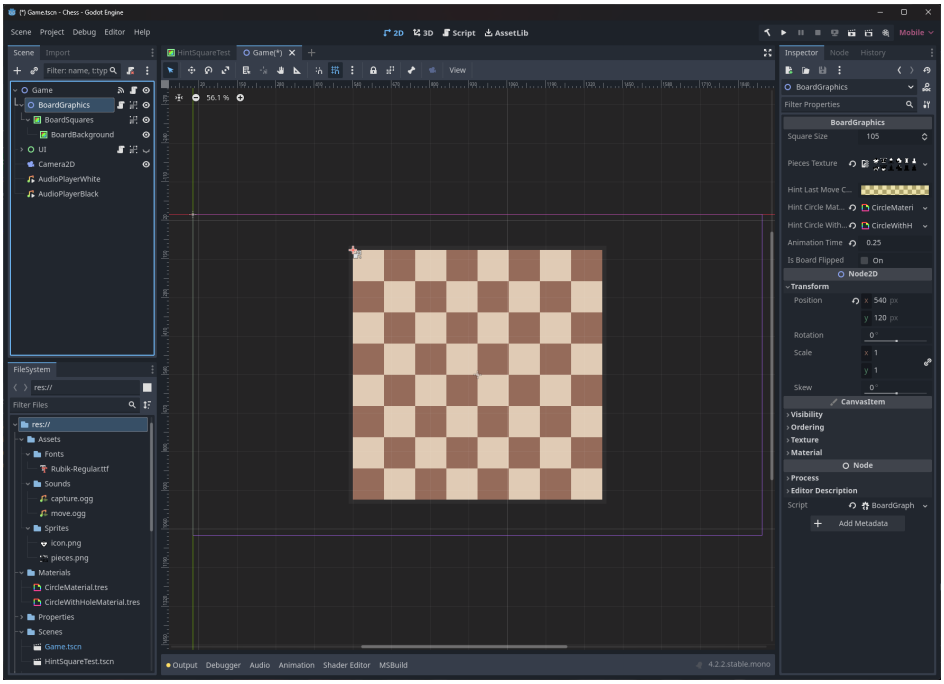

Figura 4.2: Editor de Godot con la imagen del tablero

El inspector del editor de Godot presenta la jerarquía de estos nodos, podemos ver como el nodo *Game* correspondiente a la clase *Game*, que vimos en el apartado 2.5, es la raíz del árbol de nodos y que el nodo *BoardGraphics* es hijo de este, y que a su vez tiene como hijo el nodo *sprite BoardSquares*, que contiene la imagen del tablero.

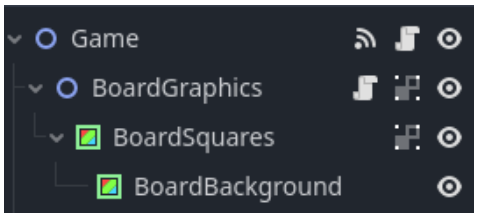

Figura 4.3: Inspector con el arbol de nodos

Las piezas

Para dibujar las piezas, contamos con un vector de *Sprite2D* de dimensión 64 al que hemos llamado *piecesSprites*. Cada uno de estos *sprites* se colocará en la posición correspondiente a la casilla del tablero a la que pertenece y en función de la pieza que se encuentre actualmente en el tablero se mostrará una imagen u otra. En el caso en el que no exista ninguna pieza en esa casilla, lo que haremos será ocultar ese *sprite*.

Para representar las piezas hemos usado una única imagen la cual contiene todos los tipos de pieza para ambos colores. Esto nos ahorra de disponer de una imagen por cada pieza y color, pero también nos introduce un problema, ya que nos impide

asignar directamente la imagen de la pieza correspondiente a su respectivo *sprite* que la representa. Lo que tendremos que hacer en ese caso es *recortar* la imagen en función de la pieza que necesitemos como veremos a continuación.

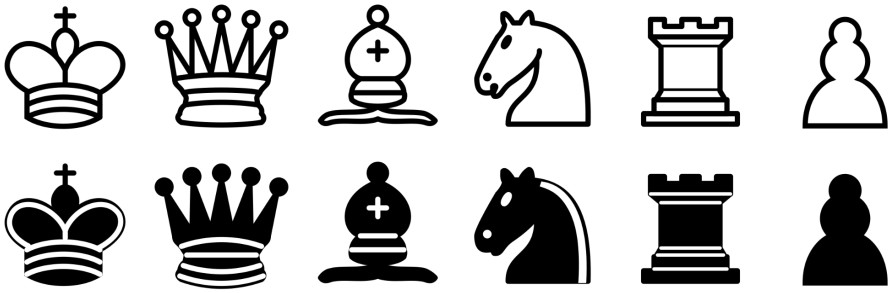

Figura 4.4: Spritesheet de las piezas

Sabiendo que la imagen completa mide en el eje x un tamaño *Width* y en el eje y un tamaño *Height* y que cada pieza tiene el mismo tamaño, para saber cuánto mide una sola pieza tenemos que dividir esa dimensión entre 6 horizontalmente y 2 verticalmente, con lo que nos quedaría que la *sub-imagen* de una sola pieza tiene dimensión:

$$\left(\frac{Width}{6}\right) \times \left(\frac{Height}{2}\right)$$

Finalmente nos faltaría calcular los *offsets* dentro de la imagen. Estos *offsets* nos indican la posición de la *sub-imagen* de una pieza específica dentro de la imagen completa. Aquí podemos ver el ejemplo si la pieza elegida hubiese sido el caballo negro.

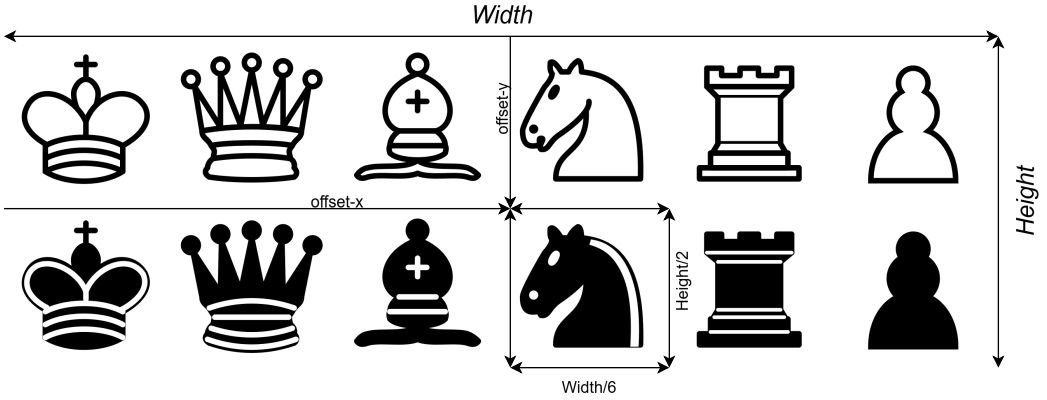

Figura 4.5: Posición y tamaño de la pieza del caballo dentro de la imagen completa

Para calcular estos *offsets* seguiríamos las siguientes fórmulas para ambos ejes de coordenadas:

$$\text{Offset}_x = n_x \times \left(\frac{\text{Width}}{6}\right), \quad \text{donde} \quad 0 \leq n_x \leq 5$$

$$\text{Offset}_y = n_y \times \left(\frac{\text{Height}}{2}\right), \quad \text{donde} \quad 0 \leq n_y \leq 1$$

Siendo n_x y n_y las coordenadas expresadas en número de piezas en la que se encuentra la *sub-imagen* que queremos representar. En el caso del ejemplo del caballo negro anterior n_x tendria el valor de 3 y n_y el valor de 1.

De realizar todo este procedimiento se ocupa la función *SetPieceSprite*, aquí podemos ver su implementación:

```
1  private void SetPieceSprite(Sprite2D pieceSprite, Piece piece)
2  {
3      if (piece.type == Piece.Type.None)
4      {
5          pieceSprite.Visible = false;
6      }
7      else
8      {
9          pieceSprite.RegionRect = new Rect2(
10             pieceTextureSize.X * ((int)piece.type - 1),
11             pieceTextureSize.Y * ((int)piece.color - 1),
12             pieceTextureSize.X, pieceTextureSize.Y
13         );
14         pieceSprite.Visible = true;
15     }
16 }
```

Fragmento de código 4.2: Función *SetPieceSprite*

Esta función toma como parámetros el *sprite* de las piezas al que queremos modificar su apariencia y la pieza que contiene el tipo y el color que queremos que tome. En el caso en el que el tipo sea *ninguno* lo que haremos será modificar la propiedad *Visible* del *sprite* a falso, con esto lo que haremos será ocultarlo y que no se dibuje en la pantalla. En caso contrario, modificaremos la propiedad *RegionRect* del *sprite*. Esta propiedad indica la región que queremos renderizar dentro de una textura o imagen. Esta propiedad es un *Rect2* (Rectangulo 2d) de Godot, cuenta con 4 parámetros: *offset* x, *offset* y, anchura y altura, que son precisamente los parámetros que hemos descrito y calculado previamente. Estos parámetros se modifican en función del tipo y color de la pieza. La variable *pieceTextureSize* es del tipo vector 2d y contiene el tamaño de la *sub-imagen* de una pieza.

Puesto que el tablero va cambiando a la vez que se van realizando movimientos necesitaremos ir actualizando la interfaz de usuario a la vez. Para ello dispondremos de la función *UpdateGraphics* que llamaremos cada vez que se realice un movimiento y se encargará de actualizar cada uno de los *sprites* del tablero.

```
public void UpdateGraphics()
{
    for (int j = 0; j < 8; j++)
    {
        for (int i = 0; i < 8; i++)
        {
            int index = i + j * 8;

            // update the pieces sprites to match the ones in the board

            Sprite2D pieceSprite = piecesSprites[index];
            Piece piece = board.GetPiece(index);

            SetPieceSprite(pieceSprite, piece);

            Vector2 pieceSpritePosition = new Vector2(i + 0.5f, j + 0.5f);
            pieceSprite.Position = pieceSpritePosition * squareSize;
        }
    }
}
```

Fragmento de código 4.3: Función *UpdateGraphics*

Esta función se encarga de actualizar cada *sprite* de las piezas con su pieza correspondiente del tablero y recolocarlo en el centro de la casilla en el caso en el que se haya desplazado.

Los indicadores

Sería útil poder indicar al jugador qué movimientos tiene disponibles una determinada pieza. Para ello dispondremos de una función que tome una lista de movimientos y represente en el tablero de forma gráfica las casillas a las cuales corresponden dichos movimientos. La función *SetHintMoves* hace precisamente esto que hemos descrito, seguiremos un procedimiento similar al que vimos con los *sprites* de las piezas, en este caso utilizamos el vector de *sprites* de dimension 64 *hintsSprites*.

```
public void SetHintMoves(List<Move> moves)
{
    // hide previous hint moves
    for (int i = 0; i < 64; i++)
    {
        hintsSprites[i].Visible = false;
    }

    // show the new ones if not null
    if (moves != null)
    {
        foreach (Move move in moves)
        {
            Material m = move.pieceTarget.type == Piece.Type.None ?
                hintCircleMaterial : hintCircleWithHoleMaterial;

```

```
17                 hintsSprites[move.squareTargetIndex].Material = m;
18                 hintsSprites[move.squareTargetIndex].Visible = true;
19             }
20         }
21 }
```

Fragmento de código 4.4: Función *SetHintMoves*

Esta función se encarga primero de ocultar todos los indicadores para luego mostrar aquellos que le hemos pasado mediante la lista de movimientos, con la diferencia de que si la casilla se encuentra vacía se elige el *sprite* de un círculo relleno y si está ocupada por una pieza se elige el *sprite* de un círculo sin relleno. Podemos ver el resultado de esto en la imagen 4.6, la cual toma como ejemplo los movimientos posibles de la reina blanca.

Figura 4.6: Ejemplo de los indicadores de movimiento

También añadiremos los indicadores que muestran el último movimiento realizado en el tablero. Para ello consultaremos el último movimiento y dibujaremos dos cuadrados amarillos, uno en la casilla en la que se encontraba la pieza antes de realizar el movimiento y otro en la casilla a la que se ha desplazado dicha pieza. Para ello haremos uso de la función *_Draw* de la clase *Node2D* que nos proporciona Godot para el dibujo de primitivas 2d, como por ejemplo en este caso cuadrados de color amarillo.

```
1 public override void _Draw()
2 {
3     // draw last move
```

```
4
5       if (board.TryGetLastMove(out Move lastMove))
6       {
7           int si = lastMove.squareSourceIndex % 8;
8           int sj = lastMove.squareSourceIndex / 8;
9           int ti = lastMove.squareTargetIndex % 8;
10          int tj = lastMove.squareTargetIndex / 8;
11
12          DrawRect(new Rect2(new Vector2(si, sj) * squareSize,
13              squareSize, squareSize), hintLastMoveColor);
14          DrawRect(new Rect2(new Vector2(ti, tj) * squareSize,
15              squareSize, squareSize), hintLastMoveColor);
16      }
17  }
```

Fragmento de código 4.5: Función *_Draw*

4.2. Integrando la interfaz de usuario en el juego

Hasta ahora todo lo que hemos ido viendo de la clase *BoardGraphics* ha sido acerca de su funcionalidad. Sin embargo, todavía nos falta integrarla en el juego junto con el resto de elementos. Además de implementar al jugador humano que se encargará de controlarla, puesto que la clase *BoardGraphics* no hace nada por sí misma, sólo expone una interfaz con la que podemos interactuar con ella, y decirle qué debe representar y cuándo se debe de actualizar.

4.2.1. El jugador humano

El jugador humano se encargará de controlar la interfaz de usuario mediante la clase *BoardGraphics*, que permitirá al usuario interactuar y realizar movimientos en el tablero. El jugador humano extenderá de la clase *Player* que vimos en el apartado 2.4 cuando hablábamos sobre los jugadores. Sobreescribiremos el método *Update* de la clase *Player* e implementaremos toda la funcionalidad para seleccionar piezas y moverlas en el tablero. Para ello, dispondremos de una máquina de estados que se encargará de todo ello.

Código de *PlayerHuman*

Antes de entrar en detalles sobre la máquina de estados, mostramos a continuación el código del jugador humano (*PlayerHuman*).

```
1 using Godot;
2 using System;
3 using System.Collections.Generic;
4
5 public class PlayerHuman : Player
```

```
6  {
7      private enum InputState
8      {
9          Idle,
10         Dragging,
11         PieceSelected
12     }
13
14     // player state
15     private InputState inputState = InputState.Idle;
16
17     // board
18     private Board board;
19
20     // board graphics
21     private BoardGraphics boardGraphics;
22
23     // piece selection helpers
24     private int pieceSelectedIndex = -1; // -1 means nothing selected
25     private List<Move> pieceSelectedMoves = null;
26
27     public PlayerHuman(Board board, BoardGraphics boardGraphics)
28     {
29         // init
30         this.board = board;
31         this.boardGraphics = boardGraphics;
32     }
33
34     public override void Update()
35     {
36         // Aqui se actualiza la maquina de estados...
37     }
38 }
```

Fragmento de código 4.6: Clase *PlayerHuman*

La función *Update* en la clase *PlayerHuman* es de hecho el núcleo del ciclo de actualización de la máquina de estados del jugador humano. Esta función se encarga de determinar el estado actual del jugador y ejecutar la lógica correspondiente a ese estado. La función *Update* hace esto mediante un *switch* que selecciona y ejecuta la función adecuada según el estado actual (*Idle*, *Dragging*, o *PieceSelected*).

```
1  public override void Update()
2  {
3      // handle everything related to move selection
4
5      // get the mouse coordinates
6      Vector2 mouse = boardGraphics.GetLocalMousePosition();
7
8      // get the square the mouse is on
9      bool isOnSquare = boardGraphics.TryGetSquareIndexFromCoords(
10         mouse,
11         out int squareIndex
12     );
13
14     // state machine
```

```
15      switch (inputState)
16      {
17          case InputState.Idle:
18              HandlePieceSelection(squareIndex, isOnSquare);
19              break;
20          case InputState.Dragging:
21              HandleDragMovement(mouse, squareIndex, isOnSquare);
22              break;
23          case InputState.PieceSelected:
24              HandleClickMovement(squareIndex, isOnSquare);
25              break;
26      }
27  }
```

Fragmento de código 4.7: Función *Update*

Antes de ejecutarse el código correspondiente a la máquina de estados, en la función *Update* se obtiene la posición actual del ratón, relativa a la posición dónde se encuentra el tablero dentro de la pantalla, y se obtiene mediante la función *TryGetSquareIndexFromCoord* en qué casilla del tablero se encuentra el ratón mediante el parámetro de salida *squareIndex*. Esta función además devuelve un valor lógico (*isOnSquare*) que indica si el ratón se encuentra dentro del tablero e indica por lo tanto la validez del parámetro *squareIndex*. Esta serie de parámetros son comunes a todos los estados de la máquina de estados, de ahi la razón por la cual los obtengamos al principio.

La máquina de estados que hemos implementado se ilustra en la Figura 4.7. Esta figura proporciona una visión clara de las transiciones entre los diferentes estados (*Idle*, *Dragging*, *PieceSelected*) y cómo se gestionan estas transiciones basadas en las interacciones del usuario. Esta máquina de estados se ocupará como hemos mencionado antes de gestionar toda la lógica que permite al usuario interactuar con el tablero y así poder realizar movimientos en este.

Estado *Idle*

El estado *Idle* se activa cuando no hay ninguna pieza seleccionada y el jugador no está realizando ninguna acción con el ratón. Si el jugador hace clic en la pantalla se verifica que se haya seleccionado una casilla dentro del tablero (*isOnSquare*) en caso de que así sea entonces se verifica si la pieza que se encuentra en esa casilla corresponde con una pieza válida (del color correspondiente al turno). En ese caso, la pieza se selecciona, guardandonos el índice de la casilla en la variable entera *pieceSelectedIndex*, se calculan, se guardan (en la variable *pieceSelectedMoves*) y se muestran sus movimientos legales (*boardGraphics.SetHintMoves*), y se cambia el estado a *Dragging*.

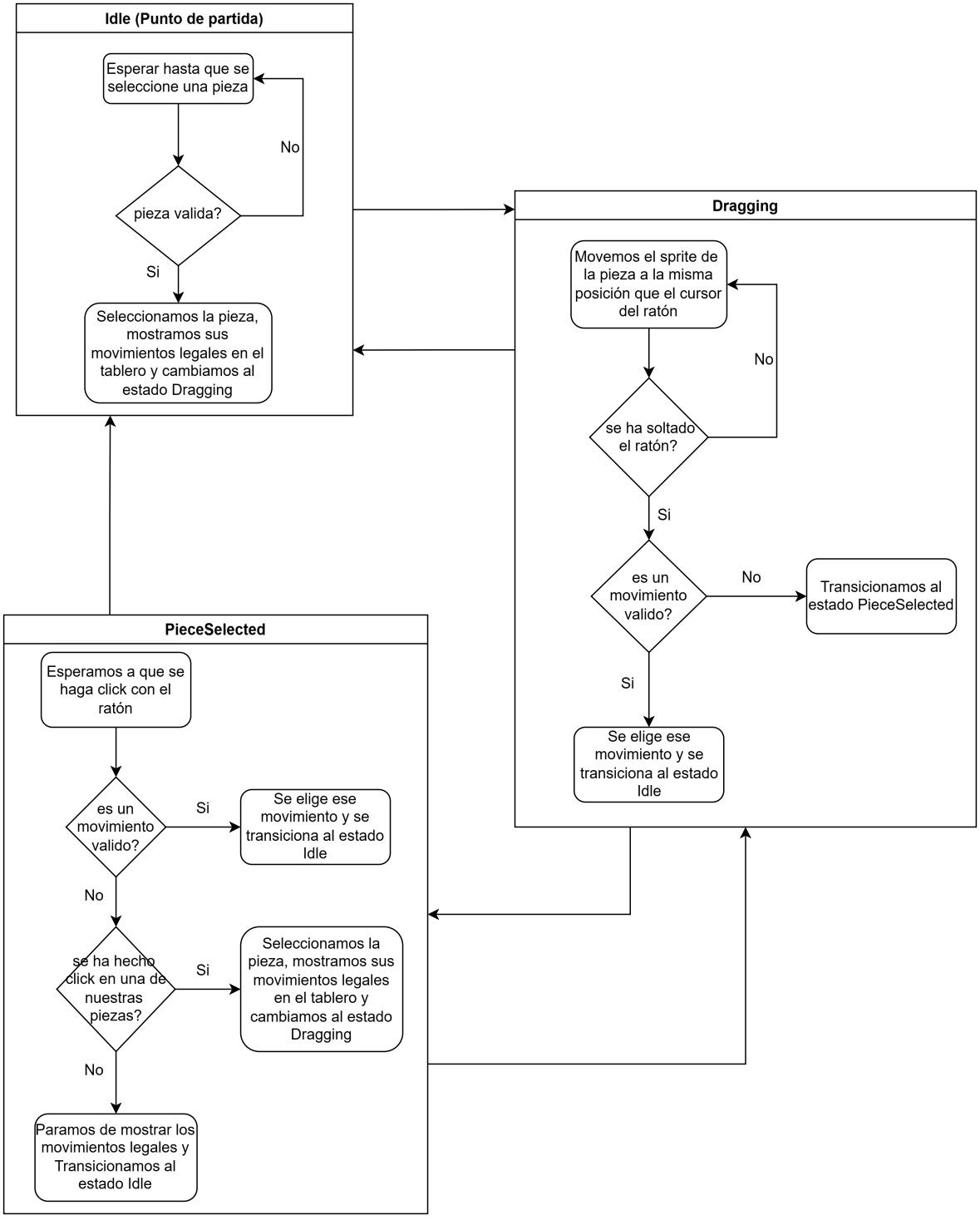

Figura 4.7: Máquina de estados del jugador humano

La función que maneja e implementa la funcionalidad de este estado es *HandlePieceSelection*.

```
1 private void HandlePieceSelection(int squareIndex, bool isOnSquare)
2 {
3     // the first frame you click
4     if (Input.IsActionJustPressed("Select"))
5     {
6         if (isOnSquare)
```

```
7           {
8               // get the piece
9               Piece piece = board.GetPiece(squareIndex);
10
11              // if not none then select it
12              if (piece.type != Piece.Type.None
13                  && piece.color == board.GetTurnColor())
14              {
15                  // select piece
16                  pieceSelectedIndex = squareIndex;
17                  pieceSelectedMoves
18                      = MoveGeneration.GetLegalMoves(board, squareIndex);
19
20                  // set hint moves
21                  boardGraphics.SetHintMoves(pieceSelectedMoves);
22
23                  // change state
24                  inputState = InputState.Dragging;
25              }
26          }
27      }
28  }
```

Fragmento de código 4.8: Función *HandlePieceSelection*

Estado *Dragging*

Este estado se activa cuando una pieza ha sido seleccionada y el jugador la está arrastrando. Mientras el ratón está presionado, el *sprite* de la pieza que está seleccionada se coloca en la posición del cursor del ratón mediante la función *SetPieceSpritePosition* que vimos en la clase *BoardGraphics*. Cuando el jugador suelta el ratón (*IsActionJustReleased("Select")*) se comprueba primero si se ha soltado en una casilla dentro del tablero (*isOnSquare*). En ese caso, se comprueba si la casilla elegida para efectuar el movimiento corresponde a uno de los movimientos legales de la pieza que tenemos seleccionada. Esto lo conseguimos probando con todos los movimientos de dicha pieza que habiamos seleccionado (almacenados en la variable *pieceSelectedMoves*) y comprobando que la casilla destino del movimiento (*move.squareTargetIndex*) corresponde con la casilla que habiamos seleccionado (*squareIndex*). En ese caso, elegimos ese movimiento disparando el evento *onMoveChosen* mediante la llamada a la función *OnMoveChosen* y pasándole como parámetro el movimiento elegido. Si recordamos cuando hablabamos sobre los jugadores y la clase *Game* (secciones 2.4 y 2.5) esto provocará que se notifique a la clase *Game* y sea ésta finalmente la que juegue el movimiento elegido en el tablero. Finalmente, el estado vuelve a *Idle*. En el caso en el que el que no se haya seleccionado una casilla en el tablero o el movimiento no sea válido, se actualizan los gráficos del tablero (esto provoca que el sprite de la pieza seleccionada vuelva a su posición original) y el estado actual cambia a *PieceSelected*.

La función que maneja e implementa la funcionalidad de este estado es *HandleDragMovement*.

```
private void HandleDragMovement(Vector2 mouse, int squareIndex, bool isOnSquare)
{
    // move the piece selected to the mouse position
    boardGraphics.SetPieceSpritePosition(pieceSelectedIndex, mouse);

    // if stop holding
    if (Input.IsActionJustReleased("Select"))
    {
        if (isOnSquare)
        {
            // check for a valid move
            foreach (Move move in pieceSelectedMoves)
            {
                if (move.squareTargetIndex == squareIndex)
                {
                    // chose the move (no animation)
                    ChoseMove(move, false);

                    // reset board state
                    inputState = InputState.Idle;

                    // move selected
                    return;
                }
            }
        }

        // update graphics
        boardGraphics.UpdateGraphics();

        // go to piece selected state
        inputState = InputState.PieceSelected;
    }
}
```

Fragmento de código 4.9: Función *HandleDragMovement*

Estado *PieceSelected*

Este estado se utiliza cuando el jugador ha seleccionado una pieza pero no la está arrastrando. El jugador puede hacer clic en una casilla para intentar mover la pieza seleccionada. Si el jugador hace clic en una casilla válida para mover la pieza, se realiza el movimiento y el estado regresa a *Idle*. Si el jugador selecciona una nueva pieza del mismo color, se actualizan los movimientos posibles y el estado cambia a *Dragging* de nuevo. Si el jugador hace clic en una casilla no valida, el estado regresa a *Idle* y se desactivan las sugerencias de movimientos pasándole *null* a la función *boardGraphics.SetHintMoves*.

La función que maneja e implementa la funcionalidad de este estado es *HandleClickMovement*.

```
private void HandleClickMovement(int squareIndex, bool isOnSquare)
{
    // the first frame you click
    if (Input.IsActionJustPressed("Select"))
    {
        if (isOnSquare)
        {
            // check for a valid move
            foreach (Move move in pieceSelectedMoves)
            {
                if (move.squareTargetIndex == squareIndex)
                {
                    // select the move
                    ChoseMove(move, true);

                    // reset board state
                    inputState = InputState.Idle;

                    // move selected
                    return;
                }
            }

            // if the move is not legal then check if another piece is selected
            // get the piece
            Piece piece = board.GetPiece(squareIndex);

            // if not none then select it
            if (piece.type != Piece.Type.None && piece.color == board.GetTurnColo
            {
                // select piece
                pieceSelectedIndex = squareIndex;
                pieceSelectedMoves = MoveGeneration.GetLegalMoves(board,  squareI

                // set hint moves
                boardGraphics.SetHintMoves(pieceSelectedMoves);

                // change state to holding the piece
                inputState = InputState.Dragging;

                // exit
                return;
            }
        }

        // disable hint moves
        boardGraphics.SetHintMoves(null);

        // go back to idle state
        inputState = InputState.Idle;
    }
}
```

Fragmento de código 4.10: Función *HandleClickMovement*

4.2.2. Integración con la clase *Game*

Finalmente nos faltaría integrar toda la interfaz de usuario con el núcleo de nuestro juego, es decir, con la clase *Game*. Conectaremos la interfaz con el tablero y la iremos actualizando, a la vez que se realizan los movimientos.

En la función *_Ready* (Código 2.8) de la clase *Game*, añadiremos la conexión de la interfaz gráfica (*BoardGraphics*) con el tablero (*Board*) usando la función *ConnectToBoard*. Luego, actualizaremos la interfaz para que los *sprites* de las piezas coincidan con las piezas del tablero. Adicionalmente, ahora que tenemos desarrollado al jugador humano podemos hacer que ambos jugadores, blanco y negro, sean del tipo *PlayerHuman* lo que nos permitirá poder jugar contra otra persona o con nosotros mismos para probar que todo lo que hemos ido programando funciona correctamente.

```
1  public override void _Ready()
2  {
3      // init the board and load the fen
4      board = new Board();
5      board.LoadFenString(Board.StartFEN);
6
7      // connect the board graphical representation with the board itself
8      boardGraphics.ConnectToBoard(board);
9
10     // update the board graphics
11     boardGraphics.UpdateGraphics();
12
13     // creamos los jugadores
14     playerWhite = new PlayerHuman(board, boardGraphics);
15     playerBlack = new PlayerHuman(board, boardGraphics);
16
17     // nos suscribimos a los eventos de ambos jugadores
18     playerWhite.onMoveChosen += OnMoveChosen;
19     playerBlack.onMoveChosen += OnMoveChosen;
20
21     // comenzamos en el estado NextTurn
22     gameState = GameState.NextTurn;
23 }
```

Fragmento de código 4.11: Conexión entre el tablero y la interfaz de usuario

En la función *OnMoveChosen* (Código 2.10), además de realizar el movimiento en el tablero, añadiremos la actualización de la interfaz de usuario.

```
1  private void OnMoveChosen(Move move)
2  {
3      // make the move
4      board.MakeMove(move);
5
6      // update ui
7      boardGraphics.SetHintMoves(null);
8      boardGraphics.UpdateGraphics();
9
```

```
10      // change state
11      gameState = GameState.NextTurn;
12  }
```

Fragmento de código 4.12: Actualización de la interfaz de usuario

Con esto concluiría el desarrollo de la interfaz de usuario, permitiendo a los jugadores interactuar de manera intuitiva y fluida con el tablero de ajedrez.

Capítulo 5

Desarrollo de la inteligencia artificial

El desarrollo de una inteligencia artificial (IA) para un juego de ajedrez es una tarea que combina conceptos avanzados de algoritmos de búsqueda, evaluación de posiciones y optimización de recursos computacionales [3]. En este capítulo, se explorará la implementación de una IA para ajedrez basada principalmente en el algoritmo minimax [17, 18, 19], y se detallarán una serie de mejoras que incrementaran su eficiencia y rendimiento.

5.1. Función de evaluación del tablero

Antes de adentrarnos en los algoritmos de búsqueda, es esencial contar con una función de evaluación del tablero que permita a la IA determinar la calidad de una posición dada de forma aproximada. Esta función se encargará de asignar un valor numérico a una posición del tablero determinada de forma estática.

Dado que cada pieza en el ajedrez es diferente provoca que no todas valgan lo mismo. Por ejemplo, una reina es mucho mas valiosa que un peón o un caballo. Por esta razón les vamos a asignar diferentes valores que reflejarán cuán valiosas son. La tabla 5.1 refleja los valores que hemos elegido y que estan basados en los valores que hemos encontrado en la *Chess Programming Wiki* [20].

Pieza	**Valor**
Peón	100
Caballo	300
Alfil	320
Torre	500
Reina	900

Tabla 5.1: Valores de las piezas de ajedrez

Al rey no lo hemos incluido puesto que no tiene valor propiamente dicho. El valor del rey no se puede cuantificar de la misma manera que las otras piezas puesto que su importancia es absoluta y esencial para la continuidad del juego.

Con respecto al posicionamiento de las piezas, es evidente que una pieza gana o pierde valor en función de la posición en la que se encuentre colocada en el tablero, con lo que tendremos que tener esto también en cuenta a la hora de valorar las piezas. Veamos un ejemplo:

En la siguiente imagen de un tablero, observamos dos caballos: uno está en una casilla central y el otro en una esquina. Esto resulta en que el caballo de la esquina controle menos casillas que el que está en el centro, lo que nos lleva a concluir que el caballo de la esquina es *menos efectivo* o no tan valioso como el del centro.

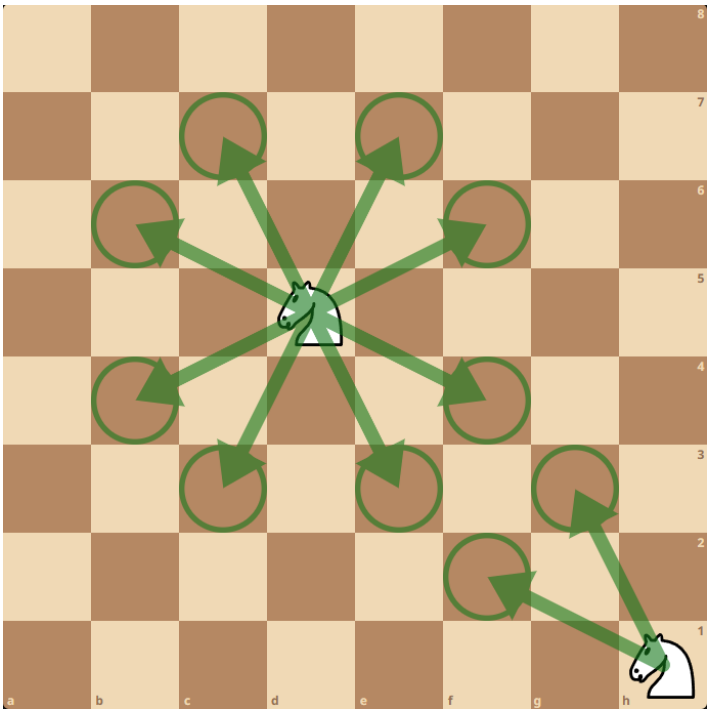

Figura 5.1: Ejemplo del valor de la posicion del caballo

Para tener en cuenta la posicion dentro del tablero para las piezas, nos vamos a crear una serie de tablas para cada tipo de pieza, las cuales añadan o resten valor en función de la casilla en la que se encuentre dicho tipo de pieza. Estas tablas están basadas en los valores que hemos obtenido de [21], y corresponden a los bonuses por casilla. Si representamos estas tablas de forma que cuanto mayor sea el bonus coloreamos el cuadrado de un azul más claro que cuanto menor sea el bonus, podemos visualizar fácilmente las mejores posiciones para cada tipo de pieza en el tablero.

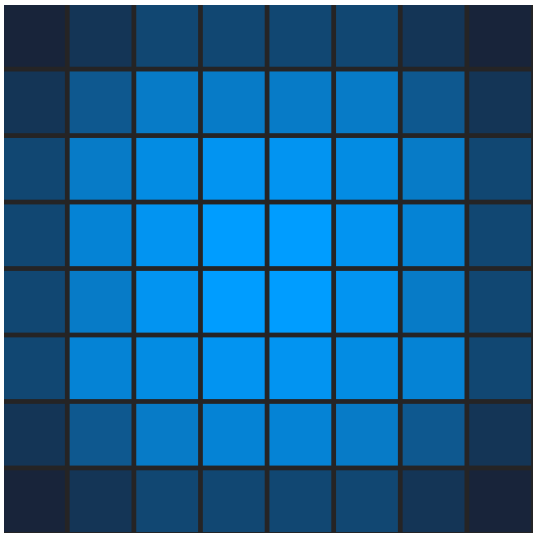

Figura 5.2: Tabla del caballo

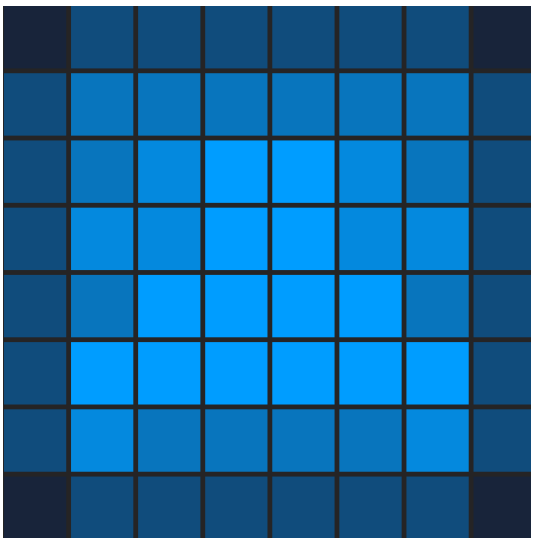

Figura 5.3: Tabla del alfil

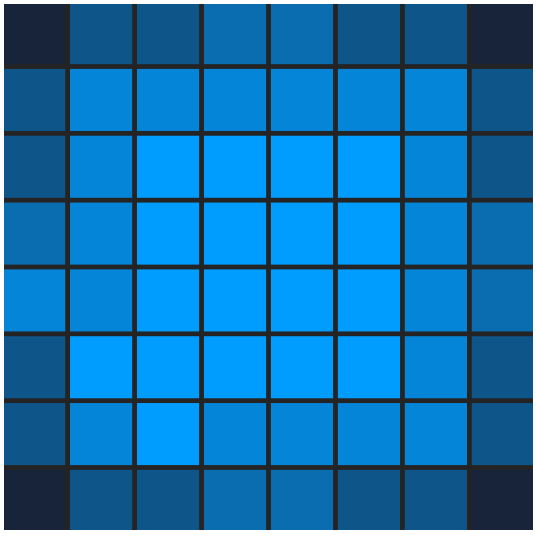

Figura 5.4: Tabla de la reina

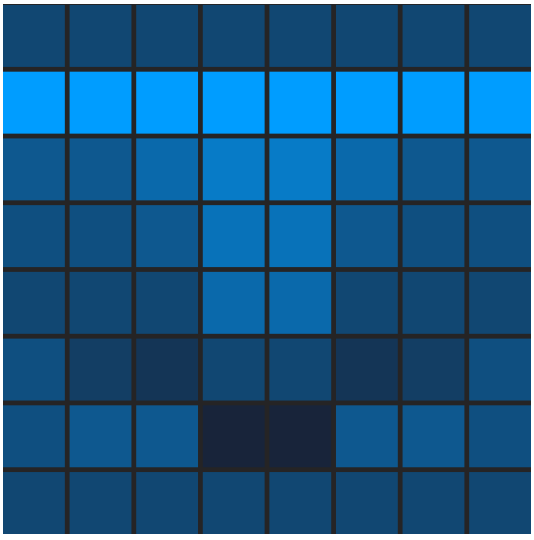

Figura 5.5: Tabla del peón

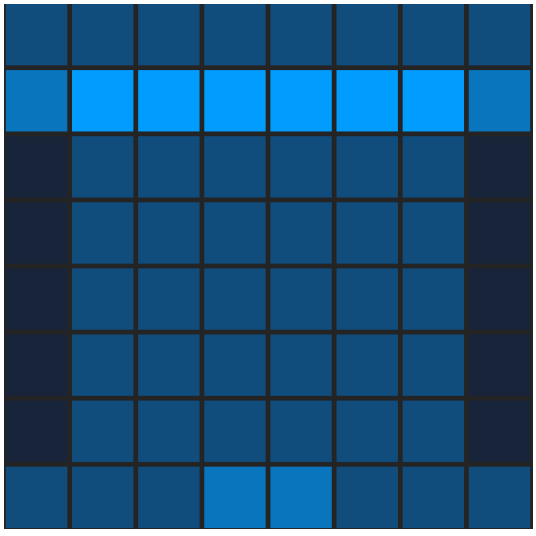

Figura 5.6: Tabla de la torre

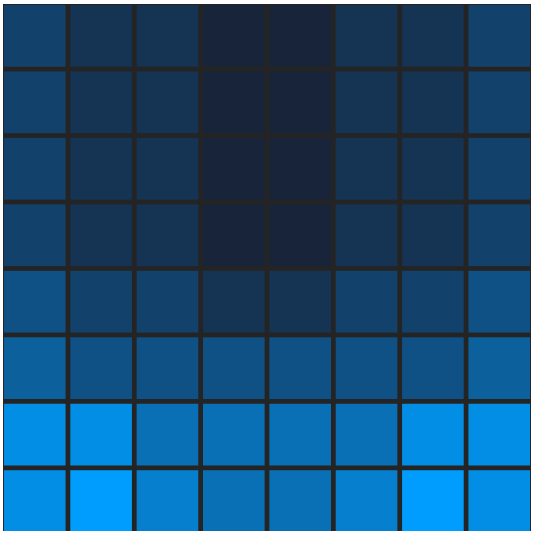

Figura 5.7: Tabla del rey

- **Caballo**: Un caballo en el centro tiene más movilidad y control sobre más casillas, por lo que es más valioso que uno en una esquina.
- **Alfil**: Los alfiles son más efectivos en diagonales largas y abiertas, por lo que su valor aumenta en posiciones donde pueden moverse libremente.
- **Reina**: Dado que la reina es la pieza más poderosa, su valor es alto en casi todas las posiciones, pero especialmente en el centro donde puede controlar más casillas.
- **Peón**: Los peones son más valiosos cuando están cerca de promocionarse, es decir, en las filas más avanzadas.
- **Torre**: Las torres son más efectivas en columnas abiertas y en la séptima fila, donde pueden atacar las piezas del oponente.
- **Rey**: La seguridad del rey es crucial, por lo que su valor aumenta en posiciones seguras y protegidas.

Las tablas que hemos visto correspondes a las tablas para las piezas blancas. Para las piezas negras se usan las mismas tablas pero rotadas 180 grados.

Para evaluar el tablero, contaremos el valor de todas las piezas blancas, sumando o restando los bonos de la casilla donde se encuentran, según las tablas previamente mencionadas. Esta será la puntuación de las blancas. Luego, repetiremos el proceso con las piezas negras para obtener su puntuación. Con ambas puntuaciones, restaremos la puntuación de las negras a la puntuación de las blancas. Una puntuación positiva indicará que las blancas se encuentran en una mejor posición, mientras que una puntuación negativa indicará que son las negras las que se encuentran en una mejor posición.

```
1  // evaluate board
2  public static int EvaluateBoard(Board board)
3  {
4      int materialWhite = CountMaterial(board, Piece.Color.White);
5      int materialBlack = CountMaterial(board, Piece.Color.Black);
6
7      return materialWhite - materialBlack;
8  }
9
10 // count material
11 private static int CountMaterial(Board board, Piece.Color color)
12 {
13     int count = 0;
14
15     foreach (int index in board.GetPiecesIndices(color))
16     {
```

```
17          Piece piece = board.GetPiece(index);
18
19          count += GetPieceValue(piece.type) +
20              PieceTables.Read(PieceTables.GetTable(piece.type), index, color);
21      }
22
23      return count;
24  }
```

Fragmento de código 5.1: Evaluación del tablero

La función *EvaluateBoard* calcula la puntuación de la posición del tablero. Para ello hace uso de la función *CountMaterial*, que se encarga de contar la puntuación del color elegido, para ello se obtienen todas las piezas de dicho color y su posición dentro del tablero y se van sumando su valor con su correspondiente bonus de forma acumulativa. La función *GetPieceValue* devuelve los valores vistos en la tabla 5.1 en función del tipo de pieza. La función *PieceTables.Read* devuelve el bonus correspondiente de la tabla de dicha pieza en función de la casilla en la que se encuentra. La función *PieceTables.GetTable* devuelve la tabla de bonuses para el tipo de pieza. Aquí podemos ver el código de ambas funciones:

```
1  // read value from a table
2  public static int Read(int[] table, int index, Piece.Color color)
3  {
4      switch (color)
5      {
6          case Piece.Color.White: return table[index];
7          case Piece.Color.Black: return table[63 - index];
8      }
9
10     return 0;
11 }
12
13 // get piece table
14 public static int[] GetTable(Piece.Type type)
15 {
16     switch (type)
17     {
18         case Piece.Type.Pawn:
19             return PawnTable;
20         case Piece.Type.Knight:
21             return KnightTable;
22         case Piece.Type.Bishop:
23             return BishopTable;
24         case Piece.Type.Rook:
25             return RookTable;
26         case Piece.Type.Queen:
27             return QueenTable;
28         case Piece.Type.King:
29             return KingTable;
30     }
31
32     return null;
```

```
33 }
```

Fragmento de código 5.2: Función *Read* y *GetTable*

5.2. Algoritmo de búsqueda: minimax

El algoritmo minimax [19] es una técnica fundamental en la teoría de juegos y la inteligencia artificial, utilizada para determinar el movimiento óptimo en juegos de dos jugadores, como el ajedrez.

5.2.1. Funcionamiento del algoritmo

El algoritmo minimax se basa en la construcción de un árbol de decisiones que representa todos los posibles movimientos, desde la posición actual hasta los estados terminales del juego. Cada nodo del árbol representa un estado del juego, y las ramas representan los movimientos posibles. Veamos su funcionamiento:

- **Generación del árbol de juego**: Se genera un árbol que incluye todos los movimientos posibles, desde la posición actual hasta los estados terminales (victoria, derrota o empate). En otros juegos, como por ejemplo el 3 en raya, es posible generar el árbol completo del juego desde una posición dada, puesto que el número de combinaciones posibles de jugadas no es muy elevado. Sin embargo, para el caso del ajedrez no es posible generar el árbol entero, puesto que estamos hablando de una cantidad extremadamente grande de combinaciones posibles. Lo que haremos será generar el árbol hasta una profundidad determinada.

- **Evaluación de nodos terminales**: Cada nodo terminal se evaluará utilizando la función de evaluación previamente desarrollada, que asigna un valor numérico a la posición.

- **Propagación de valores**: Los valores de los nodos terminales se propagan hacia arriba en el árbol. En cada nivel del árbol, los jugadores alternan entre maximizar su ganancia (jugador MAX) y minimizar la ganancia del oponente (jugador MIN).

Veamos un ejemplo:

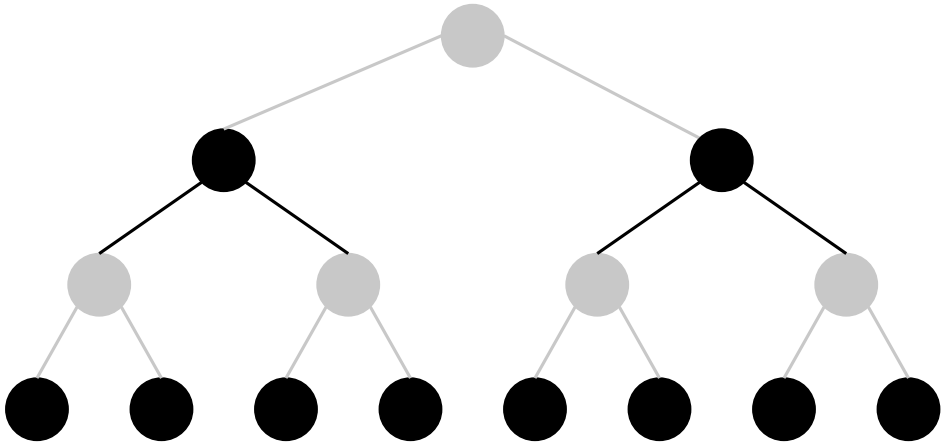

Figura 5.8: Arbol minimax 1

Para este ejemplo, imaginémonos que sólo podemos escoger entre 2 posibles movimientos para simplificar. Comenzaremos con el turno de las blancas, representado por el nodo blanco que corresponde a la raíz del árbol. Al tener dos movimientos posibles representados por las dos líneas grises, llegamos a 2 estados posibles en el que es el turno de las negras, y continuamos con el árbol un par de iteraciones más, hasta llegar a la profundidad deseada, en este caso 3, para así formar el árbol que vemos en la figura 5.8.

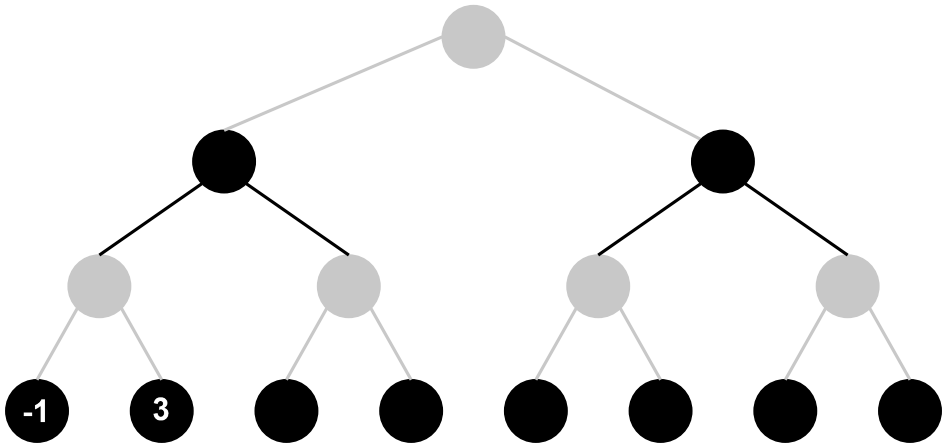

Figura 5.9: Arbol minimax 2

Una vez que hemos llegado a la profundidad deseada pasamos a evaluar las posiciones usando una función de evaluación. Imaginemos que el resultado de la función de evaluación devuelve un valor que cuanto más negativo es, mejor es la posición para las negras, mientras que cuanto más positivo es, mejor es para las blancas. Esto quiere decir que el jugador negro va a intentar *minimizar* mientras que el jugador blanco intentará *maximizar*. En el caso de nuestro ejemplo, los dos primeros nodos terminales resultan con una puntuación de -1 y 3 como podemos ver en la figura 5.9.

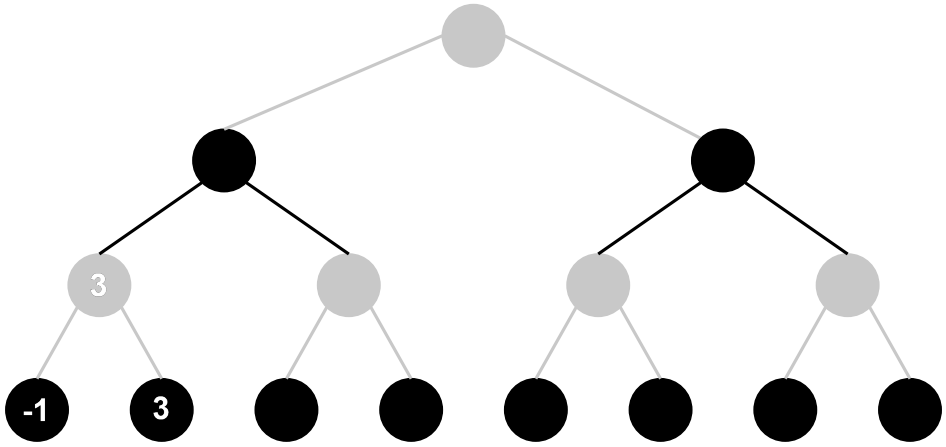

Figura 5.10: Arbol minimax 3

Una vez que tenemos evaluados ambos nodos terminales, como en el turno anterior era el turno de las blancas y estas buscan una puntuación cuanto mayor mejor, es decir, están *maximizando*, escogerán el movimiento que lleva a la puntuación de 3, como podemos ver en la figura 5.10.

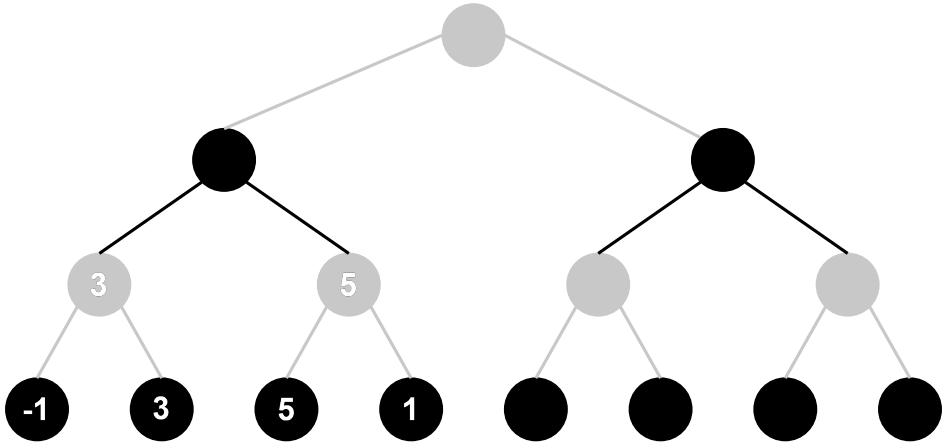

Figura 5.11: Arbol minimax 4

Repetimos lo mismo para la otra bifurcación de la primera rama del árbol.

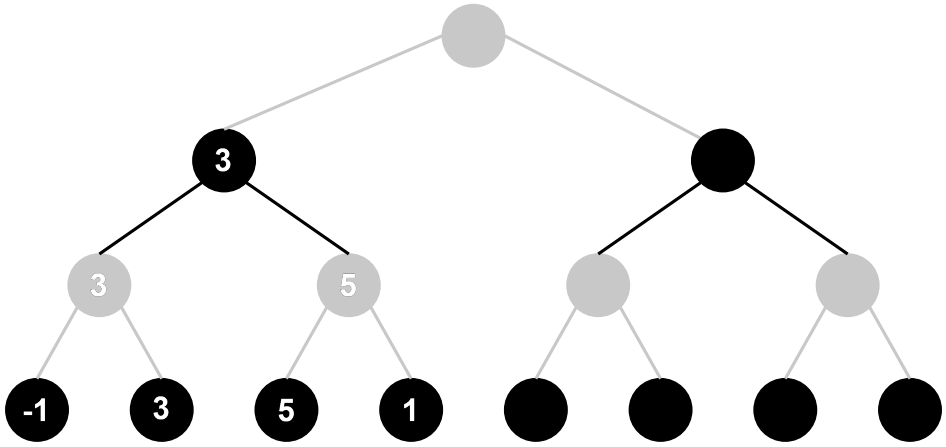

Figura 5.12: Arbol minimax 5

Ahora sería el turno de las negras, y estarían entre dos decisiones. Elegir el nodo que tiene como puntuación un 3 o el que tiene un 5. Como las negras quieren cuanta menos puntuación mejor, es decir, están *minimizando*, elegirán el movimiento que les lleva al estado de puntuación 3.

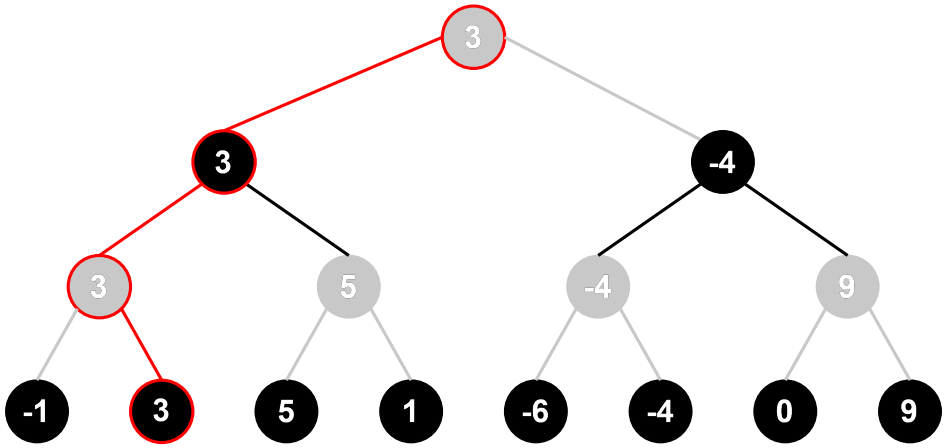

Figura 5.13: Arbol minimax 6

Repetimos este proceso con el resto del árbol, maximizando y minimizando, dependiendo del turno y podemos ver el camino seleccionado. En el nodo raíz, que sería el actual y a partir del cual se ha calculado el árbol de decisiones, las blancas (turno actual) elegirían el movimiento que les lleva por la rama de la izquierda, como podemos ver en la figura 5.13.

Aquí podemos ver la implementación en pseudocodigo del algoritmo minimax:

```
1 int maxi( int depth ) {
2     if ( depth == 0 ) return evaluate();
3     int max = -infinity;
4     for ( all moves) {
```

```
5           makeMove(move);
6           score = mini( depth - 1 );
7           undoMove();
8           if( score > max )
9               max = score;
10      }
11      return max;
12  }
13
14  int mini( int depth ) {
15      if ( depth == 0 ) return -evaluate();
16      int min = +infinity;
17      for ( all moves) {
18          makeMove(move);
19          score = maxi( depth - 1 );
20          undoMove();
21          if( score < min )
22              min = score;
23      }
24      return min;
25  }
```

Fragmento de código 5.3: Pseudocodigo minimax [19]

Se utilizan 2 subrutinas, una para el jugador que minimiza y otra para el jugador que maximiza, y se van llamando recursivamente de forma que se van alternando. Se comienza con una profundidad determinada y se va decrementando a lo largo que se va profundizando en el árbol, una vez que la profundidad llegue a 0, es decir, hasta los nodos terminales, se evalúa el tablero y se obtiene la puntuación. Si beneficia a las blancas, el jugador que maximiza, resultará en una evaluación positiva, mientras que si beneficia a las negras, el jugador que minimiza, será negativa.

Nosotros, sin embargo, vamos a implementar una variante llamada *negamax* [22, 23], que es una manera común de implementar el algoritmo minimax para juegos como el ajedrez, ya que no necesita de ambas subrutinas. Aquí podemos ver la implementación en pseudocodigo:

```
1   int negaMax( int depth ) {
2       if ( depth == 0 ) return evaluate();
3       int max = -infinity;
4       for ( all moves)  {
5           makeMove(move);
6           score = -negaMax( depth - 1 );
7           undoMove(move);
8           if( score > max )
9               max = score;
10      }
11      return max;
12  }
```

Fragmento de código 5.4: Pseudocodigo negamax [23]

Funciona de la misma manera que la implementación pura del algoritmo *minimax*, sólo que en vez de maximizar y minimizar en función del jugador, vamos a maximizar siempre y cada vez que llamemos a la función de manera recursiva negaremos la puntuación del rival. Para que esto funcione nuestra función de evaluación ya no tiene que devolver una puntuación positiva si favorece a las blancas y una negativa si favorece a las negras, tiene que devolver una puntuación relativa al jugador desde el que se está analizando el tablero, es decir, si lo estamos analizando desde la posición de las blancas una puntuación positiva significará que es bueno para las blancas y negativa que es mala para estas mismas. Igualmente si estamos viendo la posición desde el punto de vista de las negras, una puntuación positiva significará que es bueno para las negras y una negativa que es malo para estas mismas. Esto lo conseguiremos realizando la siguiente modificación en la función de evaluación (fragmento de código 5.1).

```
1 // evaluate board
2 public static int EvaluateBoard(Board board, Piece.Color colorPerspective)
3 {
4     int materialWhite = CountMaterial(board, Piece.Color.White);
5     int materialBlack = CountMaterial(board, Piece.Color.Black);
6
7     return (materialWhite - materialBlack) *
8         (colorPerspective == Piece.Color.White ? 1 : -1);
9 }
```

Fragmento de código 5.5: Modificación a la función de evaluación del tablero

5.2.2. Implementación

Comenzaremos con la implementación de la clase *Search* que es la que se encargara de realizar estas búsquedas usando el algoritmo minimax.

```
1  public class Search
2  {
3      // on search complete action
4      public event System.Action<Move> onComplete;
5
6      private Board board;
7
8      // best move found
9      private Move bestMoveFound;
10     private int bestEvalFound = negativeInfinity;
11
12     public void SetBoard(Board board)
13     {
14         this.board = board;
15     }
16
17     public void StartSearch()
18     {
```

```
19          bestMoveFound = Move.NullMove;
20          bestEvalFound = int.MinValue;
21
22          // SearchMoves(...);
23
24          onComplete?.Invoke(bestMoveFound);
25      }
26
27      public int SearchMoves(...) {...}
28  }
```

Fragmento de código 5.6: Clase *Search*

Esta clase se encargará de realizar las búsquedas sobre el tablero. Para ello se llamará a la función *StartSearch*, la cual se encargará de llamar a la función *SearchMoves*, que veremos en profundidad y es la que utilizará el algoritmo *negamax* para buscar el mejor movimiento. En cuanto acabe la búsqueda se invocará al evento *onComplete*, al cual se le pasará el mejor movimiento encontrado para esa posición y turno actuales del tablero.

```
1  public int SearchMoves(int depth, int plyFromRoot) {
2      if (depth == 0) {
3          return Evaluation.EvaluateBoard(board, board.GetTurnColor());
4      }
5
6      // check checkmate
7      List<Move> moves = MoveGeneration.GetAllLegalMovesByColor(
8          board,
9          board.GetTurnColor()
10     );
11
12     if (moves.Count == 0)
13     {
14         if (MoveGeneration.IsKingInCheck(board, board.GetTurnColor()))
15         {
16             return negativeInfinity;
17         }
18
19         // stale mate
20         return 0;
21     }
22
23     int bestEvaluation = negativeInfinity;
24
25     foreach (Move move in moves) {
26         board.MakeMove(move);
27         int evaluation = -SearchMoves(depth - 1, plyFroomRoot + 1);
28         board.UndoMove();
29
30         // check for better evaluation = better move
31         if (evaluation > bestEvaluation)
32         {
33             bestEvaluation = evaluation;
34
35             // if we are in the root (starting position) also retrieve
```

```
36              // best move
37              if (plyFroomRoot == 0)
38              {
39                  bestMove = move;
40                  bestEval = evaluation;
41              }
42          }
43      }
44
45      return bestEvaluation;
46  }
```

Fragmento de código 5.7: Función *SearchMoves*

La función *SearchMoves* toma 2 parámetros, la profundidad hasta la que queremos llegar del árbol (parámetro *depth*) y el número de *medios movimientos* que se han realizado desde la raíz del árbol (*plyFromRoot*). En ajedrez, un *ply* es un solo movimiento de un jugador. Dos *plies* corresponden a un turno completo, movimiento de las blancas seguido por un movimiento de las negras. La implementación de esta función sigue el mismo esquema que el pseudocodigo del algoritmo negamax que vimos en el fragmento 5.4, pero con una serie de añadidos, comprobaremos si estamos en posición de jaque mate o rey ahogado (cuando el número de movimientos generados en esa posición es igual a 0). En el caso de jaque mate, retornaremos una puntuacion de $-\infty$ (que nos hagan jaque mate es la peor situación que nos puede pasar), y en el caso de rey ahogado retornaremos 0 indicando tablas. Finalmente, para obtener el movimiento escogido, lo que haremos será que cada vez que se regrese, debido a la recursividad, a la raíz del árbol (*plyFromRoot* es igual a 0), es decir, al estado de partida, nos iremos guardando y reemplazando el movimiento que tenga mejor puntuación, hasta que finalice la función. Una vez que esto acabe, la variable *bestMove* contendrá el mejor movimiento y la variable *bestEval* contendrá la puntuación de éste.

Tenemos que tener en cuenta la profundidad inicial que elegimos a la hora de llamar a la función *SearchMoves* a la hora de iniciar la búsqueda, puesto que la búsqueda tardará más tiempo cuanto más profundo se intente buscar.

Una vez vista la función de búsqueda nos quedará por implementar al jugador AI (*PlayerAI*), el cual será el que se encargará de iniciar dichas búsquedas para elegir el movimiento.

```
1  public class PlayerAI : Player
2  {
3      private Board board;
4
5      private Search search;
6
```

```
7       // move selected
8       private Move moveSelected = Move.NullMove;
9       private bool moveFound = false;
10
11      // ctor
12
13      public PlayerAI(Board board)
14      {
15          // init
16
17          this.board = board;
18          search = new Search();
19          search.onComplete += OnSearchCompleted;
20      }
21
22      public override void NotifyTurnToMove()
23      {
24          moveFound = false;
25          Board boardCopy = board.Copy();
26          search.SetBoard(boardCopy);
27
28          // Start a new Task to calculate the best move asynchronously
29          Task.Run(() =>
30          {
31              search.StartSearch();
32          });
33      }
34
35      private void OnSearchCompleted(Move move)
36      {
37          moveSelected = move;
38          moveFound = true;
39      }
40
41      public override void Update()
42      {
43          if (moveFound)
44          {
45              ChoseMove(moveSelected, true);
46          }
47      }
48  }
```

Fragmento de código 5.8: Clase *PlayerAI*

La clase *PlayerAI* hereda de la clase *Player*, en el constructor creamos una instancia de la clase *Search* y nos suscribimos al evento *onComplete*, para que cuando éste se active se llame a la función *OnSearchCompleted* pasándole el movimiento. En esta función se seleccionará este movimiento y se marcará a *true* el *flag* que indica que un movimiento ha sido encontrado. Cuando al jugador AI se le notifica que es su turno, lo que hará será crear una copia del tablero actual y pasarselo a la instancia que nos habíamos creado de la clase *Search*, la razón de esta copia y no un paso por referencia es porque vamos a ejecutar la función de búsqueda de forma asincrona puesto que

la función de búsqueda "tarda tiempo.[en] completarse, y asi no bloqueamos todo el juego hasta que se complete, y de esta forma evitamos cualquier condición de carrera. Finalmente, en la función *Update* esperamos hasta que el movimiento se encuentre y cuando ésto ocurra elegimos dicho movimiento.

5.2.3. Alpha-Beta Prunning

Una optimización muy común, que se realiza en el algoritmo minimax, es aplicar *alpha-beta prunning* [18]. Ello consiste en podar ramas del árbol que no son necesarias de calcular puesto que estas no influyen en el resultado final. Veamos como aplicaríamos esta técnica al ejemplo que vimos en el apartado 5.2.1.

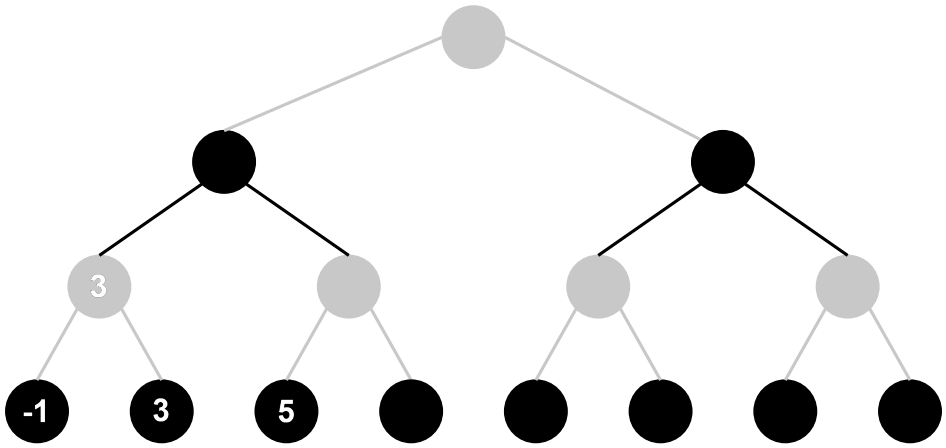

Figura 5.14: Arbol minimax (alpha-beta prunning) 1

Si regresamos al momento en el que evaluamos la posición en el tercer nodo terminal como un 5 (imagen 5.14), nos daremos cuenta de que no necesitamos calcular el segundo nodo terminal de esa misma subrama. Esto es así ya que en la raíz de esa subrama, es decir, en el nodo blanco, se tiene para elegir 2 movimientos. Pero este nodo sabe que por lo menos obtiene una puntuación de un 5, que es el nodo que acabamos de calcular. Por lo tanto, el nodo negro superior estaría decidiento entre un 3 (el del otro nodo blanco) y un 5 o más, por lo que nunca elegiria la rama del 5, puesto que preferirá la del 3 que es menor y por lo tanto no necesitamos calcular ese segundo nodo terminal.

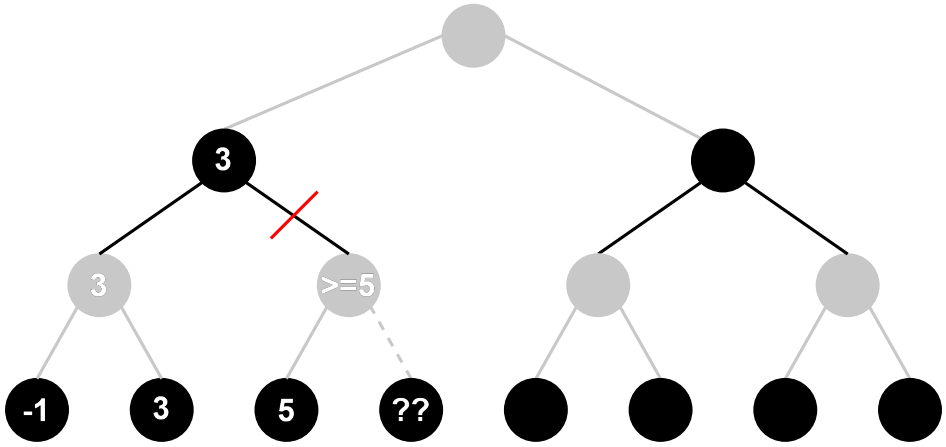

Figura 5.15: Arbol minimax (alpha-beta prunning) 2

En la imagen 5.16 podemos ver el resultado de aplicar esta optimización al ahorrarnos calcular la evaluación del nodo que hemos marcado con interrogantes (*??*).

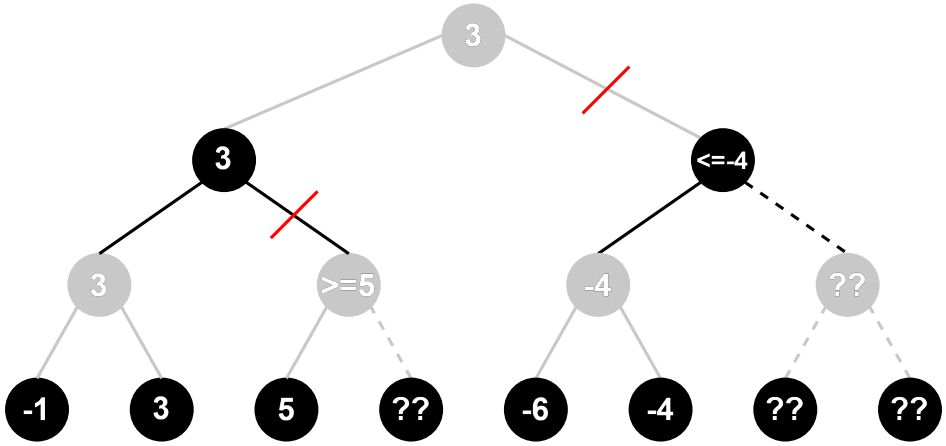

Figura 5.16: Arbol minimax (alpha-beta prunning) 3

Si continuamos con el resto de nodos faltantes podremos repetir este mismo proceso ahorrandonos más cálculos y tiempo. En este caso, sabemos que el nodo negro principal de la segunda rama tendrá una puntuación de -4 o menor, y como el nodo raíz blanco tiene como opción el nodo negro de la otra rama principal cuya puntuación es de 3, nunca elgiría la rama que estamos calculando actualmente, por lo que nos podemos ahorrar de calcular los 3 nodos faltantes.

Si modificamos nuestra función *SearchMoves* para que incluya esta optimización nos quedaria el siguiente codigo:

```
1 public int SearchMoves(int depth, int plyFromRoot, int alpha, int beta) {
2     // ...
3
4     foreach (Move move in moves) {
```

```
5           board.MakeMove(move);
6           int evaluation = -SearchMoves(depth - 1, plyFroomRoot + 1, -beta, -alpha)
7           board.UndoMove();
8
9           // hemos encontrado un movimiento muy bueno
10          // nuestro oponente seguramente no va a elegir esta rama
11          // la cortamos
12          if (evaluation >= beta)
13          {
14              return beta;
15          }
16
17          if (evaluation > alpha)
18          {
19              alpha = evaluation;
20
21              // if we are in the root (starting position) also retrieve
22              // best move
23              if (plyFroomRoot == 0)
24              {
25                  bestMove = move;
26                  bestEval = evaluation;
27              }
28          }
29      }
30
31      return alpha;
32  }
```

Fragmento de código 5.9: Función *SearchMoves*

Incluimos dos parámetros en la función que llamaremos *alpha* y *beta*. *Alpha* es el parámetro que indica nuestra mejor evaluación, mientras que *beta* es la mejor evaluación del rival. Estos parámetros se inicializarán a $-\infty$ e $+\infty$ respectivamente. Si encontramos un movimiento cuya evaluación supera a la mejor evaluación del rival ($evaluation \geq beta$), cortaremos esa rama puesto que el rival no permitirá que ese camino sea explorado, porque tiene una mejor opción. Como estamos usando la variante *negamax* cada vez que se llame de forma recursiva a la función *SearchMoves* tendremos que invertir y cambiar de signo los valores de *alpha* y *beta*, puesto que en la siguiente iteración sera el turno del rival.

Ordenación de movimientos

La idea detrás de *alfa-beta prunning* es eliminar ramas del árbol de búsqueda que no necesitan ser exploradas porque ya se ha encontrado un mejor movimiento. Si los mejores movimientos se evalúan primero, es más probable que se realicen cortes (podas) más temprano en la búsqueda, evitando así la necesidad de evaluar muchas mas ramas. Analicemos el caso del ejemplo anterior:

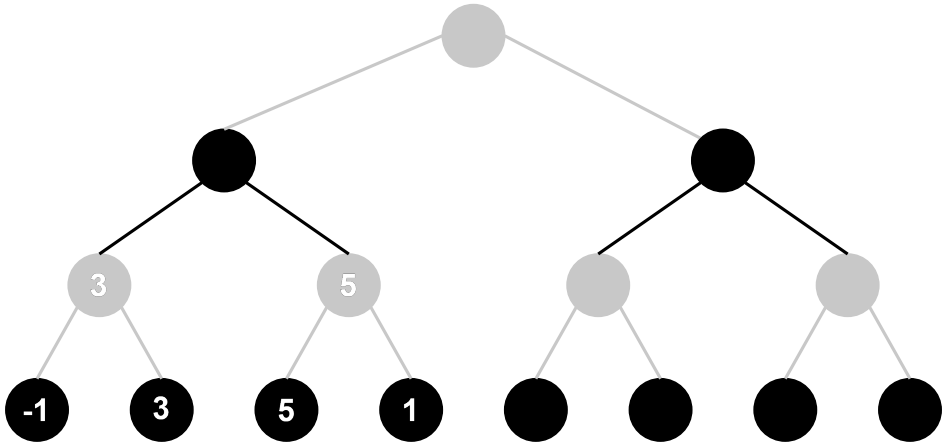

Figura 5.17: Arbol minimax (alpha-beta prunning) 3

Aquí observamos que no era necesario calcular el nodo terminal con una puntuación de 1, ya que las negras, al saber que las blancas pueden obtener una puntuación de 5 o más, nunca optarían por esa rama. Esto se debe a que se evaluó primero el nodo con una puntuación de 5, de haber sido al contrario, es decir, si hubiéramos evaluado primero el nodo con una puntuación de 1, no habríamos podido efectuar este corte como podemos ver en la imagen 5.19.

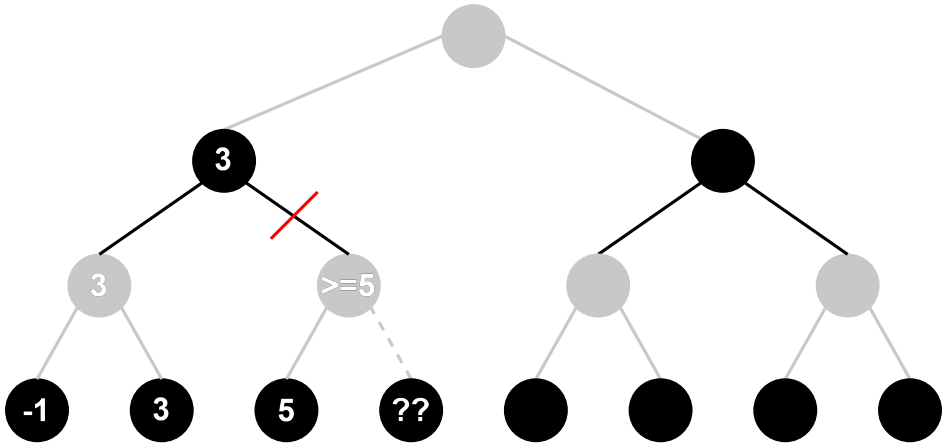

Figura 5.18: Arbol minimax (alpha-beta prunning) 3

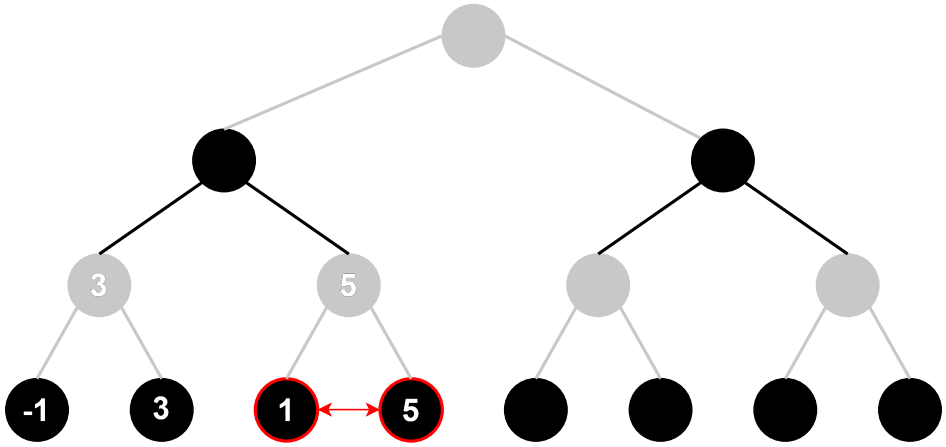

Figura 5.19: Arbol minimax, no se ha podido realizar el corte

Inicialmente, no podemos determinar si un movimiento tendrá un resultado positivo o negativo. No obstante, podemos hacer suposiciones para prever cuáles podrían ser los movimientos más acertados y así ordenarlos antes de examinarlos en la búsqueda. Algunos de los criterios que podemos considerar para evaluar si un movimiento es prometedor incluyen:

- **Desarrollo de piezas**: Movimientos que desarrollan piezas hacia posiciones activas y útiles.
- **Capturas**: Capturar una pieza de mucho valor del rival con una de poco valor.
- **Promociones de peones**.

La función *SortMoves* se encarga de ordenar una lista de movimientos en función de si creemos que son buenos o malos para un determinado color basado en los parámetros que hemos mencionados.

```
1  private void SortMoves(List<Move> moves, Piece.Color color)
2  {
3      int[] moveScore = new int[moves.Count];
4
5      for (int i = 0; i < moves.Count; i++)
6      {
7          moveScore[i] = 0;
8
9          // check if it is a capture
10
11         Piece.Type pieceTypeTarget = moves[i].pieceTarget.type;
12         Piece.Type pieceTypeSource = moves[i].pieceSource.type;
13
14         if (pieceTypeTarget != Piece.Type.None)
15         {
16             // bonus for capture
17
```

```
18              moveScore[i] += 10 * Evaluation.GetPieceValue(pieceTypeTarget) - Evalu
19          }
20          else
21          {
22              // bonus for moving the piece into a better square
23
24              int[] table = PieceTables.GetTable(pieceTypeSource);
25              moveScore[i] += PieceTables.Read(table, moves[i].squareTargetIndex, co
26          }
27
28          // bonus for promotion
29
30          if (moves[i].flags == Move.Flags.Promotion)
31          {
32              moveScore[i] += Evaluation.GetPieceValue(moves[i].promotionPieceType)
33          }
34      }
35
36      // Sort the moves list based on the move scores array ...
37  }
```

Fragmento de código 5.10: Función *SortMoves*

Finalmente, antes de proceder con la búsqueda ordenaremos los movimientos:

```
1  public int SearchMoves(int depth, int plyFromRoot, int alpha, int beta) {
2      // ...
3
4      SortMoves(moves, board.GetTurnColor());
5
6      foreach (Move move in moves) {
7          // ...
8      }
9
10     // ...
11 }
```

Fragmento de código 5.11: Función *SearchMoves*

Con la búsqueda implementada, y sus diversas optimizaciones, que hemos ido explicando y desarrollando a lo largo de este capitulo, si probamos a jugar contra el oponente manejado por la inteligencia artificial nos daremos cuenta de que está malinterpretando muchas jugadas a lo largo de la partida, las cuales acaban en malos resultados. Esto se debe a que cuando se finaliza la búsqueda en una profundidad determinada, es posible que estemos evaluando el tablero en una posición *inestable*, es decir, que en el tablero todavía existan movimientos tacticos como capturas, lo cual provoca que nuestra función de evaluación malinterprete la posición dando resultados incorrectos. Veamos un ejemplo:

Imaginemos que durante la búsqueda en el turno de las negras hemos llegado a la profundidad deseada - 1, lo cual resulta en la posición de la figura 5.20. Ahora las negras tienen que encontrar su mejor movimiento y se procederá a la evaluación de

cada una de las posiciones resultantes.

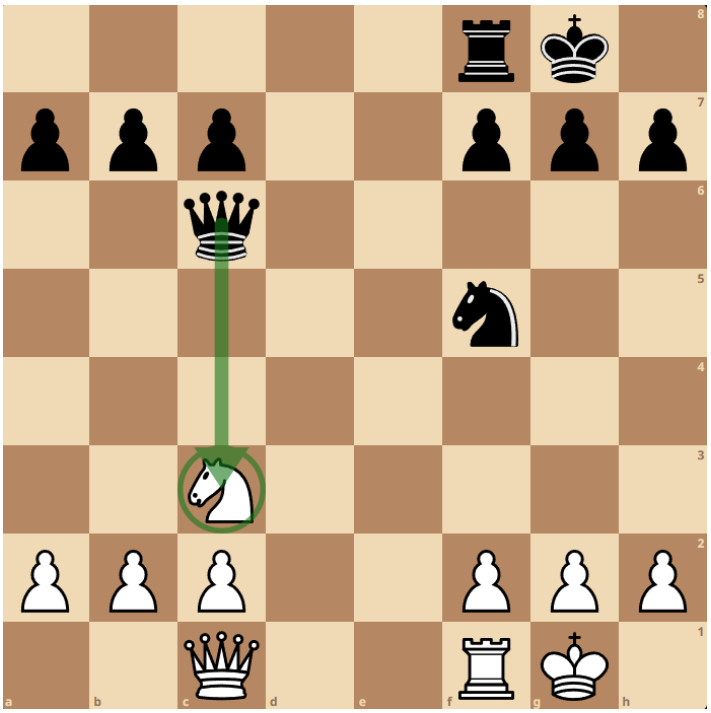

Figura 5.20: Posición FEN 5rk1/ppp2ppp/2q5/5n2/8/2N5/PPP2PPP/2Q2RK1 b - -

Las negras exploran todos los movimientos posibles y evalúan los tableros, concluyendo que, si capturan el caballo blanco con la reina negra, ganarían un caballo, ya que la función de evaluación examina el tablero estáticamente y detecta una ventaja material. No obstante, no están realmente ganando un caballo, terminarán perdiendo una reina. Este problema es el que está provocando que la inteligencia artificial tome decisiones equivocadas, esto es lo que se denomina *efecto horizonte* [24]. Para *mitigar* este efecto y si queremos que nuestra función de evaluación sea lo mas precisa posible, tendremos que tener en cuenta los movimientos tácticos siguientes.

5.3. Búsqueda de estabilidad (*Quiescence search*)

La búsqueda de estabilidad, o *quiescence search* [25], es una extensión del algoritmo minimax que se implementa para mejorar la precisión en la evaluación de posiciones complejas o tácticamente inestables. La idea principal detrás de esta técnica es continuar la búsqueda en aquellas posiciones donde el algoritmo principal podría tomar decisiones incorrectas debido a evaluaciones estáticas que no consideran las posibles secuencias tácticas inmediatas.

El objetivo de la búsqueda de estabilidad es evitar los *errores de horizonte*, donde la evaluación se detiene antes de que puedan resolverse movimientos tácticos importantes,

como capturas, jaques, o amenazas de promoción, que es precisamente el problema que nos encontramos al final de la sección anterior. Estos errores ocurren porque la evaluación estática de la posición no tiene en cuenta los cambios dinámicos que pueden ocurrir en los siguientes movimientos.

Para ello en vez de que una vez terminada la búsqueda se evalúe la posición del tablero, lo que haremos sera comenzar una nueva búsqueda la cual solo analice los movimientos tácticos, es decir, capturas de piezas.

La función *QuiescenceSearch* será la que se encargará de realizar esta nueva búsqueda que sólo se enfoca en las capturas.

```
1  private int QuiescenceSearch(int alpha, int beta)
2  {
3      // evaluate board
4      int evaluation = Evaluation.EvaluateBoard(board, board.GetTurnColor());
5
6      if (evaluation >= beta) // Beta cutoff
7      {
8          return beta;
9      }
10     alpha = Math.Max(alpha, evaluation); // Update alpha
11
12     // get moves just captures
13     List<Move> moves = MoveGeneration.GetAllLegalMovesByColor(
14         board,
15         board.GetTurnColor(),
16         true
17     );
18
19     // sort
20     SortMoves(moves, board.GetTurnColor(), Move.NullMove);
21
22     foreach (Move move in moves)
23     {
24         board.MakeMove(move, true);
25         evaluation = -QuiescenceSearch(-beta, -alpha);
26         board.UndoMove(true);
27
28         if (evaluation >= beta) // Beta cutoff
29         {
30             return beta;
31         }
32
33         alpha = Math.Max(alpha, evaluation); // Update alpha
34     }
35
36     return alpha;
37 }
```

Fragmento de código 5.12: Función *SearchMoves*

Se han realizado modificaciónes a las funciones que se encargan de generar los movimientos para que retornen solo los movimientos de capturas si el valor del flag

onlyCaptures de la función *MoveGeneration.GetAllLegalMovesByColor* es *true*. El código sigue el mismo esquema que vimos con respecto a la búsqueda normal que realizaba la función *SearchMoves*. Finalmente en la función *SearchMoves* en vez de realizar la evaluación cuando se alcanza el limite de profundidad, se llamará a la nueva búsqueda de estabilidad.

```
1  public int SearchMoves(int depth, int plyFromRoot, int alpha, int beta)
2  {
3      if (depth == 0)
4      {
5          int result = QuiescenceSearch(alpha, beta);
6          return result;
7      }
8
9      // ...
10 }
```

Fragmento de código 5.13: Función *SearchMoves*

La implementación de la búsqueda de estabilidad (*quiescence search*) nos ha permitido mejorar notablemente la precisión de las evaluaciones realizadas por la inteligencia artificial, especialmente en posiciones complejas y tácticamente inestables. Al extender la búsqueda más allá del límite de profundidad tradicional para considerar movimientos tácticos críticos, como las capturas, se pueden evitar y mitigar los errores de horizonte y tomar decisiones más sólidas.

5.4. Tabla de transposición (*Transposition table)*

Cuando estamos realizando la búsqueda, es muy probable que diferentes secuencias de movimientos conduzcan a la misma posición del tablero. Esto sucede porque en el ajedrez, las piezas pueden moverse de manera que, aunque el orden de los movimientos sea diferente, el resultado final sea exactamente el mismo.

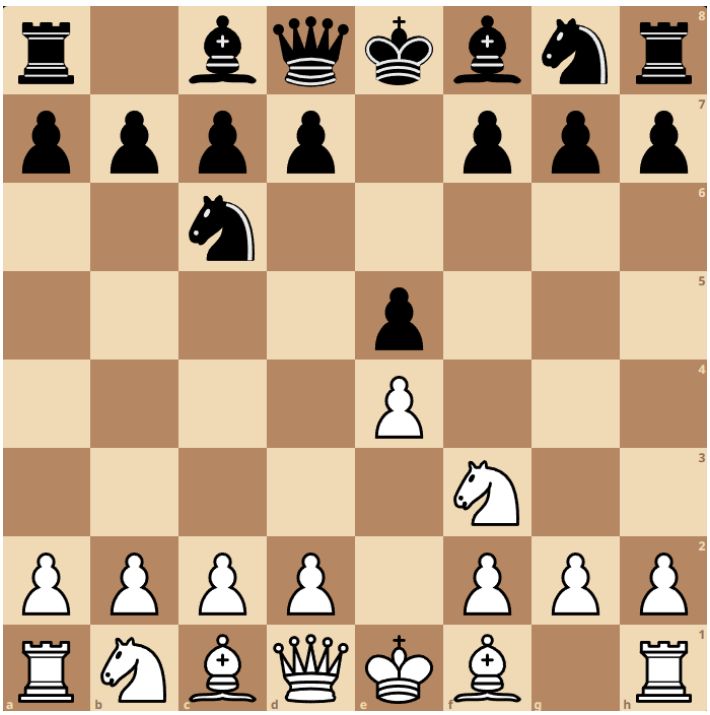

Figura 5.21: Posición FEN r1bqkbnr/pppp1ppp/2n5/4p3/4P3/5N2/PPPP1PPP/RNBQKB1R w KQkq - 4 1

Por ejemplo, para alcanzar la posición de la figura 5.21 podemos mover primero ambos peones y luego ambos caballos, que sería equivalente a comenzar moviendo el caballo blanco, luego ambos peones y por último el caballo negro.

Estas posiciones equivalentes, llamadas transposiciones [26], pueden surgir repetidamente durante la exploración del árbol de búsqueda. Por lo que es ineficiente recalcular la evaluación de posiciones que ya han sido analizadas. Aquí es donde las tablas de transposición [27] resultan esenciales.

La tabla de transposición es una base de datos que almacena los resultados de búsquedas previas, con lo que si nos encontramos con la misma posición y ya se encuentra calculada en dicha tabla, nos limitaremos a consultarla. Con ello nos ahorramos recalcular la búsqueda para dicha posición ganando tiempo. Cada entrada de la tabla almacena una serie de valores que son fundamentales para el proceso de búsqueda y evaluación. Estos valores incluyen:

- **Clave Zobrist (Zobrist Key)**.
- **Valor de la Evaluación (Evaluation Value)**: Valor numérico que representa la evaluación de la posición.
- **Profundidad de Búsqueda (Search Depth)**: Número de movimientos explorados desde la posición actual.

- **Mejor Movimiento (Best Move)**: El movimiento que resultó en la mejor evaluación desde la posición almacenada.

- **Tipo de Nodo (Node Type)**: En el contexto de la búsqueda en árboles con el algoritmo alfa-beta, cada entrada en la tabla de transposición puede estar asociada con un tipo de nodo que describe cómo debe interpretarse el valor almacenado para esa posición. Estos tipos de nodo proporcionan información sobre el estado de la evaluación y cómo se debe utilizar esa información en la búsqueda. Los tipos de nodo se definen como sigue:

  - **Exact (Nodo Exacto)**:
    - **Descripción**: Un nodo marcado como `Exact` indica que el valor almacenado en la tabla de transposición es la evaluación exacta para esa posición. La búsqueda ha sido completamente realizada en esa posición y el valor almacenado representa la evaluación precisa para esa profundidad de búsqueda.
  - **LowerBound (Límite Inferior)**:
    - **Descripción**: Un nodo marcado como `LowerBound` indica que el valor almacenado es un límite inferior para la posición. Esto significa que el valor real de la posición es al menos el valor almacenado, pero podría ser mayor. Este tipo de nodo es el resultado de una poda beta (es decir, la búsqueda se cortó porque el valor almacenado es suficientemente alto para ser un límite inferior).
  - **UpperBound (Límite Superior)**:
    - **Descripción**: Un nodo marcado como `UpperBound` indica que el valor almacenado es un límite superior para la posición. Esto significa que el valor real de la posición es a lo sumo el valor almacenado, pero podría ser menor. Este tipo de nodo es el resultado de una poda alfa (es decir, la búsqueda se cortó porque el valor almacenado es suficientemente bajo para ser un límite superior).

Aquí podemos ver el código con los valores que almacena una entrada de la tabla de transposición:

```
1 public enum NodeType
2 {
3     Exact,  // Exact score
4     LowerBound, // Lower bound (beta cut-off)
5     UpperBound // Upper bound (alpha cut-off)
```

```
}

public struct Entry
{
    public ulong key;
    public byte depth;
    public int value;
    public NodeType nodeType;
    public Move move;
}
```

Fragmento de código 5.14: Entrada de la tabla de transposición

Y aquí podemos ver parte del código de la tabla de transposición [28]:

```
public class TranspositionTable
{
    // lookup failed value
    public const int lookupFailed = int.MinValue;

    private Entry[] entries;

    public TranspositionTable(int size)
    {
        entries = new Entry[size];
    }

    public Entry GetEntry(ulong key)
    {
        int index = (int)(key % (ulong)entries.Length);
        return entries[index];
    }

    public void Store(ulong key, int depth, int value, NodeType nodeType,
        Move move)
    {
        // replace scheme
        int index = (int)(key % (ulong)entries.Length);

        entries[index] = new Entry()
        {
            key = key,
            depth = (byte)depth,
            value = value,
            nodeType = nodeType,
            move = move
        };
    }

    public int Lookup(ulong key, int depth, int alpha, int beta) {// ...}
}
```

Fragmento de código 5.15: Clase *TranspositionTable*

Para almacenar estas entradas en la tabla de transposiciones, utilizaremos un vector (*entries*) donde cada posición representa una entrada potencial de la tabla.

El tamaño del vector dependerá de la cantidad de posiciones que deseamos almacenar simultáneamente teniendo en cuenta también la memoria que éste ocupará.

Para indexar las posiciones del tablero de forma eficiente dentro de este vector, usamos la clave *zobrist*. Pero puesto que esta clave tiene un tamaño de 64 bits y no podemos tener un vector de dimension 2^{64}, lo que haremos será dado un tamaño fijo del vector realizaremos la operación modulo del *hash zobrist* con el tamaño del vector, dándonos así el índice correspondiente.

$$\text{index} = \text{zobrist} \quad \text{mód size}$$

Esto provoca que aparezca un problema y es que puesto que estamos haciendo la operación modulo, puede pasar que para 2 claves *zobrist* diferentes resulten en el mismo índice del vector lo que provocaría una colisión a la hora de almacenar o consultar posiciones (nos podria llegar a pasar que se consulta una posición y se recibe la evaluación de otra diferente). La probabilidad de que ocurran estas colisiones dependerá del tamaño de nuestro vector. Ante esto no podemos realmente hacer nada y optaremos por usar un esquema de reemplazamiento, es decir, cada vez que alamacenemos una posición, ignoraremos si ya está siendo ocupada y la reescribiremos como podemos ver en la función *Store*.

Aquí podemos ver en detalle el código de la función *Lookup*, la cual se encargará de la búsqueda de la entrada dada por la clave *zobrist* especificada.

```
1  public int Lookup(ulong key, int depth, int alpha, int beta)
2  {
3      int index = (int)(key % (ulong)entries.Length);
4      Entry entry = entries[index];
5
6      if (entry.key == key)
7      {
8          if (entry.depth >= depth)
9          {
10             switch (entry.nodeType)
11             {
12                 case NodeType.Exact:
13                     return entry.value;
14                 case NodeType.LowerBound:
15                     if (entry.value >= beta)
16                     {
17                         return entry.value;
18                     }
19                     break;
20                 case NodeType.UpperBound:
21                     if (entry.value <= alpha)
22                     {
23                         return entry.value;
24                     }
25                     break;
26             }
```

```
27          }
28      }
29
30      return lookupFailed;
31  }
```

Fragmento de código 5.16: Función *Lookup*

Si se encuentra la entrada dentro de la tabla, se comprueba que la profundidad para la cual se evaluó previamente es como minimo igual a la que se está realizando en la consulta actualmente. Esto se debe a que no nos interesan evaluaciones que se hicieron a una menor profundidad a la que estamos buscando, puesto que esos resultados se considerarán desactualizados.

Si la profundidad es válida, entonces se comprueba el tipo de nodo:

- **Nodo Exacto (NodeType.Exact)**: Si el tipo de nodo es *Exact*, esto indica que el valor almacenado es la evaluación exacta de la posición. En este caso, se devuelve directamente el valor almacenado, ya que representa la evaluación precisa de la posición en cuestión.
- **Límite Inferior (NodeType.LowerBound)**: Si el tipo de nodo es *LowerBound*, esto significa que el valor almacenado es un límite inferior. Esto ocurre cuando la búsqueda previa determinó que el valor de la posición es al menos tan bueno como el valor almacenado, pero no se pudo determinar con certeza si era mejor. Si este valor es mayor o igual al umbral *beta* actual, entonces se devuelve, lo que indica que la posición es lo suficientemente fuerte como para cortar la búsqueda en esta rama (lo que se conoce como poda beta). De lo contrario, se continúa la búsqueda, ya que el valor almacenado no es lo suficientemente alto como para tomar una decisión definitiva.
- **Límite Superior (NodeType.UpperBound)**: Si el tipo de nodo es *UpperBound*, esto significa que el valor almacenado es un límite superior. Esto ocurre cuando la búsqueda previa determinó que el valor de la posición es a lo sumo tan malo como el valor almacenado, pero podría ser peor. Si este valor es menor o igual al umbral *alpha* actual, entonces se devuelve, lo que sugiere que la posición es lo suficientemente débil como para cortar la búsqueda en esta rama (lo que se conoce como poda alfa). Si el valor es mayor que *alpha*, se continúa la búsqueda, ya que la posición podría ser mejor de lo que sugiere este límite superior.

Si ninguna de estas condiciones se cumple, la función *Lookup* concluye que la información almacenada no es suficiente para tomar una decisión en la búsqueda

actual, y por lo tanto, devuelve un valor que indica que la búsqueda debe continuar (*lookupFailed*). Este enfoque garantiza que sólo se utilicen evaluaciones de posiciones que sean relevantes y precisas para la búsqueda en curso, optimizando así el proceso de búsqueda.

Finalmente modificaremos nuestra función de búsqueda para que incluya el uso de la tabla de transposiciones:

```
public int SearchMoves(int depth, int plyFromRoot, int alpha, int beta)
{
    // get zobristKey from board
    ulong zobristKey = board.GetZobristKey();

    // check the transposition table
    int ttVal = tt.Lookup(zobristKey, depth, alpha, beta);
    if (ttVal != TranspositionTable.lookupFailed)
    {
        if (plyFromRoot == 0)
        {
            TranspositionTable.Entry tEntry = tt.GetEntry(zobristKey);
            bestMove = tEntry.move;
            bestEval = tEntry.value;
        }

        return ttVal;
    }

    // when reached 0 depth perform a quiescence search
    if (depth == 0)
    {
        int result = QuiescenceSearch(alpha, beta);
        return result;
    }

    // check for checkmate
    List<Move> moves = MoveGeneration.GetAllLegalMovesByColor(
        board,
        board.GetTurnColor()
    );

    if (moves.Count == 0)
    {
        if (MoveGeneration.IsKingInCheck(board, board.GetTurnColor()))
        {
            int result = -mateScore + plyFromRoot;
            return result;
        }

        // stale mate

        return 0;
    }

    // sort moves
    SortMoves(moves, board.GetTurnColor(), hashMove);
```

```
48
49      // calculate eval
50      NodeType nodeType = NodeType.UpperBound;
51      Move bestMoveInThisPosition = Move.NullMove;
52
53      for (int i = 0; i < moves.Count; i++)
54      {
55          board.MakeMove(moves[i]);
56          int evaluation = -SearchMoves(depth - 1, plyFromRoot + 1, -beta, -alpha);
57          board.UndoMove();
58
59          if (evaluation >= beta) // Beta cutoff
60          {
61              tt.Store(zobristKey, depth, beta, NodeType.LowerBound, moves[i]);
62              return beta;
63          }
64
65          if (evaluation > alpha)
66          {
67              nodeType = NodeType.Exact;
68              bestMoveInThisPosition = moves[i];
69
70              alpha = evaluation;
71
72              if (plyFromRoot == 0)
73              {
74                  bestMove = moves[i];
75                  bestEval = evaluation;
76              }
77          }
78      }
79
80      tt.Store(zobristKey, depth, alpha, nodeType, bestMoveInThisPosition);
81
82      return alpha;
83  }
```

Fragmento de código 5.17: Función *Lookup*

Al inicio de la búsqueda de cada nodo comprobamos si la posición ya se ha evaluado y se encuentra guardada dentro de la tabla de transposiciones, de ser asi se retorna la evaluación guardada. En el caso adicional de que además nos encontremos en la raíz del árbol (*plyFromRoot == 0*) antes de retornar la evaluación, también obtendremos el mejor movimiento que habíamos guardado en la tabla para dicha entrada.

A la hora de almacenar las posiciones dentro de la tabla, es fundamental determinar el tipo de nodo (*NodeType*) al que pertenece la evaluación calculada. Esta clasificación es crucial para la correcta interpretación y reutilización de los valores almacenados en futuras búsquedas.

El tipo de nodo se determina según los valores de *alpha*, *beta* y la evaluación (*evaluation*) obtenida al explorar los movimientos legales desde la posición actual:

- **Nodo Exacto (NodeType.Exact)**: Si durante la búsqueda se encuentra un

movimiento que mejora el valor *alpha*, este se actualiza y el nodo se clasifica como *Exact*. Esto indica que la evaluación es exacta para esta posición dado que se exploraron todas las ramas críticas y se encontró un valor que podría ser el resultado final si se juega de manera óptima. El movimiento correspondiente se guarda como el mejor movimiento encontrado hasta el momento.

- **Límite Inferior (NodeType.LowerBound)**: Si la evaluación de un movimiento supera el valor *beta*, se produce una poda beta (*beta cutoff*). Esto significa que la posición es tan favorable para el jugador que el oponente evitará esta línea de juego. En este caso, el nodo se clasifica como un *LowerBound*, ya que la evaluación es un límite inferior del valor real, y se guarda este valor en la tabla de transposición.

- **Límite Superior (NodeType.UpperBound)**: Si ninguno de los movimientos explorados mejora el valor *alpha*, la evaluación de la posición queda por debajo del umbral *alpha*. En este caso, el nodo se clasifica como *UpperBound*, indicando que la posición es desfavorable, y el valor almacenado (*alpha*) es un límite superior del valor real.

Una vez determinado el tipo de nodo, se almacena en la tabla de transposición junto con la evaluación, la profundidad y el mejor movimiento encontrado.

Todo este proceso optimiza la búsqueda en el árbol de movimientos al evitar reevaluar posiciones ya exploradas y permite cortar ramas del árbol que no conducirán a un mejor resultado, mejorando así la eficiencia de nuestra inteligencia artificial

5.5. Profundización iterativa (*Iterative deepening*)

Hasta este punto, hemos discutido la búsqueda de movimientos en un árbol de decisiones con una profundidad fija. Este enfoque tiene la ventaja de ser predecible en términos de recursos computacionales: sabemos exactamente cuántos niveles del árbol se explorarán y podemos estimar el tiempo que tomará completar la búsqueda. Sin embargo, surge una pregunta fundamental: ¿a qué profundidad deberíamos realizar la búsqueda?

La respuesta ideal sería "lo más profundo posible", ya que explorar más niveles en el árbol de decisiones generalmente lleva a evaluaciones más precisas y movimientos de mayor calidad. Cuanto más profundamente podamos buscar, mejor podremos anticipar las consecuencias de cada movimiento, lo que debería resultar en una estrategia de juego más fuerte.

No obstante, esta profundidad de búsqueda está limitada por un recurso esencial, el tiempo. A medida que la profundidad aumenta, el número de nodos que el algoritmo debe explorar crece exponencialmente, lo que incrementa considerablemente el tiempo necesario para completar la búsqueda. En un juego de ajedrez, donde las decisiones deben tomarse en un tiempo limitado, este aumento de tiempo puede volverse impracticable.

Por lo tanto, debemos encontrar un equilibrio entre la profundidad de la búsqueda y el tiempo para realizarla. Esto nos lleva a la necesidad de un enfoque que permita explorar tan profundamente como sea posible dentro de los límites de tiempo disponibles, maximizando así la calidad de la búsqueda sin exceder los recursos computacionales. Aquí es donde entra en juego la técnica de *iterative deepening*.

La idea principal es realizar múltiples búsquedas consecutivas, aumentando gradualmente la profundidad en cada iteración hasta que se alcance un límite de tiempo o profundidad predefinido. En otras palabras, se explora el árbol de decisiones a una profundidad inicial de 1, luego de 2, luego de 3, y así sucesivamente, hasta que se agote el tiempo disponible o se alcance la profundidad máxima deseada. Esto puede parecer bastante ineficiente puesto que estamos repitiendo todo el trabajo cada vez que realizamos una búsqueda a una profundidad mayor por cada iteración que realizamos. Sin embargo, gracias al uso de la tabla de transposiciones es muy probable que ya hayamos evaluado las posiciones en la anterior búsqueda. Además, podemos usar el mejor movimiento que encontramos en la búsqueda anterior y ordenarlo para que se coloque el primero en ser analizado. Aunque la función de búsqueda no siempre estará de acuerdo con que este movimiento es el mejor, muchas veces sí que lo hará, y gracias a la poda alfa-beta podremos cortar muchas ramas, con lo que paradójicamente la búsqueda iterativa es más rápida que la búsqueda normal.

Vamo a modificar la función *SortMoves* para que si le pasamos un movimiento como parámetro al que llamaremos *hashMove*, si lo encuentra en la lista de movimientos le ponga una puntuación muy grande garantizando así que sea colocado el primero.

```
1  private void SortMoves(List<Move> moves, Piece.Color color, Move hashMove)
2  {
3      int[] moveScore = new int[moves.Count];
4
5      for (int i = 0; i < moves.Count; i++)
6      {
7          moveScore[i] = 0;
8
9          // check if is the hash move
10         if (moves[i].IsEqual(hashMove))
11         {
12             // bonus for hash move
13             moveScore[i] += 10000000;
```

```
14                 continue;
15             }
16
17             // ...
18         }
19 }
```

Fragmento de código 5.18: Función *StartSearch*

En la función *SearchMoves*, debemos realizar varias modificaciones para adaptarla a la técnica de *iterative deepening*. A continuación, se detallan los cambios necesarios:

- **Ordenación de movimientos:** Al ordenar los movimientos, es importante determinar si nos encontramos en la raíz del árbol de búsqueda. Si es así, pasaremos como parámetro *hashMove* el mejor movimiento encontrado en la iteración anterior, es decir, la variable *bestMove*. Si no estamos en la raíz del árbol, utilizaremos el mejor movimiento almacenado en la tabla de transposición para la posición actual.

- **Mejor movimiento por iteración:** Debido a que ahora realizamos búsquedas iterativas, la función de búsqueda no encontrará el mejor movimiento de forma global (representado por la variable *bestMove*). En su lugar, identificaremos el mejor movimiento en cada iteración específica, almacenándolo en la variable *bestMoveIteration*.

- **Cancelación de la búsqueda:** Dado que en *iterative deepening* es posible cancelar la búsqueda en cualquier momento, por ejemplo, si se agota el tiempo, es necesario comprobar si la búsqueda ha sido cancelada mediante el indicador *isSearchCanceled*. Si se detecta que la búsqueda ha sido cancelada, la función retornará 0, provocando la cancelación de todas las búsquedas recursivas en curso.

Aqui podemos ver el código producto de dichas modificaciones.

```
1  public int SearchMoves(int depth, int plyFromRoot, int alpha, int beta)
2  {
3      // if search canceled
4      if (isSearchCanceled)
5      {
6          return 0;
7      }
8
9      // get zobristKey from board
10     ulong zobristKey = board.GetZobristKey();
11
12     // check the transposition table
13     int ttVal = tt.Lookup(zobristKey, depth, alpha, beta);
14     if (ttVal != TranspositionTable.lookupFailed)
```

```
15      {
16          if (plyFromRoot == 0)
17          {
18              TranspositionTable.Entry tEntry = tt.GetEntry(zobristKey);
19              bestMoveIteration = tEntry.move;
20              bestEvalIteration = tEntry.value;
21          }
22
23          return ttVal;
24      }
25
26      // when reached 0 depth perform a quiescence search until a stable state (to
27
28      if (depth == 0)
29      {
30          int result = QuiescenceSearch(alpha, beta);
31          return result;
32      }
33
34      // check checkmate
35
36      List<Move> moves = MoveGeneration.GetAllLegalMovesByColor(
37          board,
38          board.GetTurnColor()
39      );
40
41      if (moves.Count == 0)
42      {
43          if (MoveGeneration.IsKingInCheck(board, board.GetTurnColor()))
44          {
45              int result = -mateScore + plyFromRoot;
46              return result;
47          }
48
49          // stale mate
50
51          return 0;
52      }
53
54      // sort moves
55      Move hashMove;
56
57      if (plyFromRoot == 0)
58      {
59          hashMove = bestMoveFound;
60      }
61      else
62      {
63          TranspositionTable.Entry entry = tt.GetEntry(zobristKey);
64          hashMove = entry.move;
65      }
66
67      SortMoves(moves, board.GetTurnColor(), hashMove);
68
69      // calculate eval
70
71      NodeType nodeType = NodeType.UpperBound;
72      Move bestMoveInThisPosition = Move.NullMove;
```

```
73
74      for (int i = 0; i < moves.Count; i++)
75      {
76          board.MakeMove(moves[i]);
77          int evaluation = -SearchMoves(depth - 1, plyFromRoot + 1, -beta, -alpha);
78          board.UndoMove();
79
80          // if search canceled
81          if (isSearchCanceled)
82          {
83              return 0;
84          }
85
86          if (evaluation >= beta) // Beta cutoff
87          {
88              tt.Store(zobristKey, depth, beta, NodeType.LowerBound, moves[i]);
89              return beta;
90          }
91
92          if (evaluation > alpha)
93          {
94              nodeType = NodeType.Exact;
95              bestMoveInThisPosition = moves[i];
96
97              alpha = evaluation;
98
99              if (plyFromRoot == 0)
100             {
101                 bestMoveIteration = moves[i];
102                 bestEvalIteration = evaluation;
103             }
104         }
105     }
106
107     tt.Store(zobristKey, depth, alpha, nodeType, bestMoveInThisPosition);
108
109     return alpha;
110 }
```

En la función *StartSearch*, en lugar de buscar a una profundidad fija, realizaremos una búsqueda iterativa desde la profundidad 1 hasta la 100. En cada iteración, actualizaremos el mejor movimiento con el hallado en la iteración anterior, hasta que la búsqueda sea cancelada o se alcance el límite de profundidad. Crearemos una función denominada *CancelSearch* que se encargará de cambiar el indicador *isSearchCanceled* a *true* indicando asi que se ha cancelado la búsqueda.

```
1 public void Cancel()
2 {
3     isSearchCanceled = true;
4 }
5
6 public void StartSearch()
7 {
8     // prepare the search
```

```
9
10      bestMoveFound = Move.NullMove;
11      bestEvalFound = int.MinValue;
12      isSearchCanceled = false;
13
14      // iterative deepening
15
16      for (int depth = 1; depth < 100; depth++)
17      {
18          bestMoveIteration = Move.NullMove;
19          bestEvalIteration = negativeInfinity;
20
21          SearchMoves(depth, 0, negativeInfinity, positiveInfinity);
22
23          if (!bestMoveIteration.IsEqual(Move.NullMove))
24          {
25              bestMoveFound = bestMoveIteration;
26              bestEvalFound = bestEvalIteration;
27          }
28      }
29
30      onComplete?.Invoke(bestMoveFound);
31  }
```

Fragmento de código 5.19: Funciones *StartSearch* y *Cancel*

Finalmente, cuando se notifica al jugador IA de que es su turno, además de crear la tarea para iniciar la búsqueda, generará una tarea adicional que, tras un tiempo determinado, cancelará dicha búsqueda.

```
1  public override void NotifyTurnToMove()
2  {
3      moveFound = false;
4      Board boardCopy = board.Copy();
5      search.SetBoard(boardCopy);
6
7      // Start a new Task to calculate the best move asynchronously
8
9      Task.Run(() =>
10     {
11         search.StartSearch();
12     });
13
14     // cancelamos la busqueda pasado x tiempo
15
16     Task.Delay(searchTime).ContinueWith((t) =>
17     {
18         search.Cancel();
19     });
20 }
```

Fragmento de código 5.20: Función *NotifyTurnToMove()*

Para conocer la profundidad a la que se esta buscando podemos mostrar por consola la profundidad de la actual iteración y otros parámetros, como por ejemplo, el mejor

movimiento, la evaluación de la posición y el tiempo acumulado de la busqueda ademas de indicar a que profundidad se cancelo la búsqueda.

```
Stopwatch stopwatch = new Stopwatch();

//...

stopwatch.Start();
SearchMoves(depth, 0, negativeInfinity, positiveInfinity);
stopwatch.Stop();

if (!bestMoveIteration.IsEqual(Move.NullMove))
{
    bestMoveFound = bestMoveIteration;
    bestEvalFound = bestEvalIteration;
}

if (isSearchCanceled)
{
    GD.Print($"Search canceled at depth: {depth}");
    GD.Print($"
        Partial search result best move:{Utils.FromMoveToString(bestMoveFound)},
        eval: {bestEvalFound},
        time: {stopwatch.ElapsedMilliseconds} ms"
    );
    break;
}
else
{
    GD.Print(
        $"Depth: {depth},
        best move: {Utils.FromMoveToString(bestMoveFound)},
        eval: {bestEvalFound}, time: {stopwatch.ElapsedMilliseconds} ms"
    );
}

// ...
```

Fragmento de código 5.21: Estadisticas de la busqueda

En la imagen 5.22 tenemos un ejemplo de la salida por consola. Para este ejemplo se ha utilizado un tiempo de búsqueda de 3 segundos y se ha realizado el movimiento e2e4, a lo que la inteligencia artificial después de llegar a una profundiad de 12 ha concluido que la mejor respuesta es realizar el movimiento e7e5, que da como resultado una evaluación para las negras de -15:

```
Depth: 1, best move: b8c6, eval: 10, time: 1 ms
Depth: 2, best move: b8c6, eval: -40, time: 2 ms
Depth: 3, best move: b8c6, eval: 10, time: 7 ms
Depth: 4, best move: g8f6, eval: -35, time: 25 ms
Depth: 5, best move: b8c6, eval: 0, time: 52 ms
Depth: 6, best move: b8c6, eval: -35, time: 112 ms
Depth: 7, best move: b8c6, eval: 0, time: 214 ms
Depth: 8, best move: b8c6, eval: -25, time: 282 ms
Depth: 9, best move: b8c6, eval: -20, time: 472 ms
Depth: 10, best move: e7e5, eval: -25, time: 1343 ms
Depth: 11, best move: e7e5, eval: -15, time: 2397 ms
Search canceled at depth: 12
Partial search result best move: e7e5, eval: -15, time: 2997 ms
```

Figura 5.22: Ejemplo de salida por consola

5.6. Reducciones por traslado tardío (*Late move reductions*)

Hasta ahora, durante la búsqueda, a pesar de que ordenamos los movimientos desde el más prometedor hasta el menos prometedor, estamos analizando todos ellos a la misma profundidad. Sin embargo, sería más eficiente si los movimientos que consideramos menos relevantes se evaluaran a una profundidad menor. A esta técnica se le denomina *late move reductions*(*LMR*) [29].

Existen ciertas condiciones que deben cumplirse para que un movimiento sea considerado menos relevante y, por lo tanto, evaluado a una profundidad reducida. Estas condiciones típicamente incluyen:

- **Movimientos no principales:** Los primeros movimientos en la lista, que se consideran más prometedores, se evalúan a la profundidad completa. LMR se aplica a movimientos que aparecen después en la lista, que se consideran menos prometedores.

- **No capturas ni promociones:** LMR normalmente se aplica a movimientos "silenciosos", es decir, aquellos que no son capturas o jaques. Los movimientos que podrían cambiar drásticamente la evaluación de la posición, como capturas, no se reducen en profundidad.

- **Profundidad suficiente:** La técnica de LMR se suele aplicar sólo cuando la profundidad restante de la búsqueda es mayor que un umbral determinado, por ejemplo, 2 o más. Si la profundidad es muy baja, no se aplica la reducción.

En el caso de que al evaluar un movimiento a profundidad reducida, su puntuación supere la mejor evaluación actual, se procederá a rehacer la búsqueda a la profundidad completa, puesto que esto indicaría que el movimiento es más bueno de lo que realmente parecía al realizar la ordenación de movimientos.

Veamos la modificación que hemos realizado en la función *SearchMoves* para añadir este cambio:

```
1  //...
2  for (int i = 0; i < moves.Count; i++)
3  {
4      board.MakeMove(moves[i]);
5
6      int evaluation = 0;
7      bool needsFullSearch = true;
8      bool isCapture = moves[i].pieceTarget.type != Piece.Type.None;
9      bool isInCheck = MoveGeneration.IsKingInCheck(board, board.GetTurnColor());
10
11     // apply late move reduction if the conditions are met
12     if (i >= 3 && depth > 3 && !isCapture && !isInCheck)
13     {
14         const int reduction = 2; // incremented to 2
15         evaluation = -SearchMoves(depth - 1 - reduction, plyFromRoot + 1, -beta,
16         needsFullSearch = evaluation > alpha;
17     }
18
19     if (needsFullSearch)
20     {
21         evaluation = -SearchMoves(depth - 1, plyFromRoot + 1, -beta, -alpha);
22     }
23
24     board.UndoMove();
25
26     //...
27 }
```

Fragmento de código 5.22: Modificación añadida a la función *SearchMoves*

Solo aplicaremos LMR si se cumplen las condiciones mencionadas anteriormente, en el caso de que estas se cumplan, reduciremos la profundidad del movimiento en 2.

La técnica LMR ofrece una mejora significativa en la eficiencia al reducir la profundidad de evaluación de movimientos menos prometedores. Esto permite explorar en profundidad los movimientos que son más propensos a ser buenos, optimizando así el tiempo de cálculo y mejorando la calidad general de la búsqueda. Sin embargo, es crucial aplicar LMR con cuidado, asegurando que sólo se reduzcan aquellos movimientos que realmente son menos relevantes, evitando así pérdidas de precisión en la evaluación. La implementación de LMR, como se muestra en la función *SearchMoves*, refleja esta estrategia, permitiendo reevaluar a profundidad completa cualquier movimiento que demuestre un potencial inesperado durante la búsqueda reducida, es decir, cuando se cumpla que $evaluation > alpha$.

Capítulo 6

Pruebas y conclusiones

A lo largo del desarrollo del proyecto, hemos asumido que el código implementado es funcional y cumple con los requisitos establecidos. No obstante, a lo largo del desarrollo se han llevado a cabo una serie de pruebas para garantizar su correcto funcionamiento y validar nuestras suposiciones.

Una de las pruebas más significativas que realizamos fue la evaluación de la generación de movimientos, puesto que es la funcionalidad donde más errores aparecen y más difíciles son de encontrar. Para este propósito, implementamos un *Perft Test* [30] (*performance test*) con el objetivo de verificar dicha generación. Este test se basa en, dada una posición específica del juego, comprobar cuántos movimientos válidos y únicos pueden generarse desde esa posición. La idea principal es comparar el número de nodos generados por el sistema con el número esperado, que se conoce a priori o se puede calcular teóricamente.

Aquí podemos ver una tabla de la posición inicial que llega hasta la profundidad 7 con el número de posiciones para cada profundidad (valores obtenidos de [30]).

Depth	Nodes
0	1
1	20
2	400
3	8,902
4	197,281
5	4,865,609
6	119,060,324
7	3,195,901,860

Tabla 6.1: Número de nodos con respecto a la profundidad de la posición inicial

Para realizar este test contamos con la función *TestPositions*, que se encargará de contar el número de posiciones o nodos resultantes para una profundidad determinada.

```
1 public static ulong TestPositions(Board board, Piece.Color color, int depth)
```

```
{
    if (depth == 0)
    {
        return 1;
    }

    List<Move> moves = GetAllPseudoLegalMovesByColor(board, color);

    ulong numPositions = 0;

    foreach (Move move in moves)
    {
        board.MakeMove(move, true);
        if (!IsKingInCheck(board, color))
            numPositions += TestPositions(
                board, Piece.GetOppositeColor(color), depth - 1);
        board.UndoMove(true);
    }


    return numPositions;
}
```

Fragmento de código 6.1: Función *TestPositions*

Ahora, para realizar el test, tendremos que llamar en bucle a la función *TestPositions* hasta la profundidad que deseemos comprobar y asegurarnos de que los valores obtenidos cuadran con los teóricos.

```
private static void PerfTest()
{
    // test code

    Board newBoard = new Board();
    newBoard.LoadFenString(Board.StartFEN);
    ulong[] testNodes = new ulong[] {
        20, 400, 8902, 197281, 4865609, 119060324, 3195901860
    };

    for (int depth = 1; depth <= testNodes.Length; depth++)
    {
        ulong nodes = MoveGeneration.TestPositions(
            newBoard, Piece.Color.White, depth);

        string output = $"Depth {depth}, nodes {nodes} ";
        output += (nodes == testNodes[depth - 1]) ? "tick" :
            $"cross (expected {testNodes[depth - 1]})";

        Console.WriteLine(output);
    }

    Console.WriteLine("test finished");
}
```

Fragmento de código 6.2: Función *PerfTest*

Comprobando la salida, vemos que concuerdan con los valores teóricos, lo que nos permite constatar que la generación de movimientos funciona correctamente.

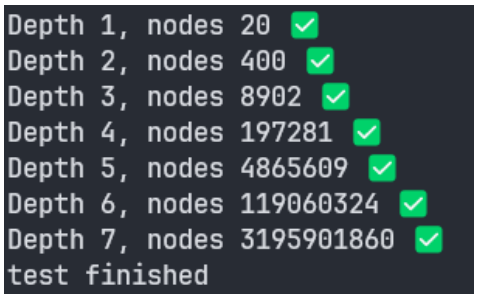

Figura 6.1: Salida en consola del test de profundidad para la posición de inicio

6.1. Estimación del ELO

El sistema de puntuación ELO [31] es un método matemático utilizado para calcular la habilidad relativa de los jugadores en disciplinas como el ajedrez. Cuanto mayor es su ELO, mejor es un jugador. Aquí podemos ver una tabla de cómo se clasificarían los rangos de ELO en el ajedrez.

Categoría	**Rango de Elo**
Principiantes	Menos de 1200
Clase D	1200 - 1399
Clase C	1400 - 1599
Clase B	1600 - 1799
Clase A	1800 - 1999
Expertos	2000 - 2199
Maestros	2200 - 2399
Grandes Maestros	2400 y superior

Tabla 6.2: Distribución de ELO en ajedrez

Para estimar el *rating* ELO de nuestro bot de ajedrez, hemos llevado a cabo un proceso metódico que implicó jugar múltiples partidas contra el motor de ajedrez Stockfish [4] a diferentes niveles de habilidad. A continuación, se detalla el proceso seguido, incluyendo las condiciones específicas bajo las cuales se realizaron las partidas.

6.1.1. Preparación del experimento

Se configuró un entorno de pruebas controlado para permitir que nuestro bot y el motor Stockfish jugaran entre sí. Para ello, tuvimos que conectar nuestro bot a Stockfish vía el protocolo *UCI* [32, 33], por el cual enviábamos los movimientos que realizaba nuestro bot y recogíamos el movimiento con el que respondía Stockfish.

6.1.2. Ejecución de las partidas

- **Número de partidas:** Nuestro bot jugó un total de 100 partidas contra Stockfish en cada nivel de ELO seleccionado (900 partidas en total). Este número de partidas fue elegido para proporcionar una muestra suficientemente grande y estadísticamente significativa de resultados.

- **Condiciones de juego:**

  - **Tiempo de pensamiento:** Ambos motores tuvieron 100 milisegundos para pensar en cada movimiento. Este límite de tiempo se aplicó para garantizar una evaluación justa del rendimiento de ambos motores bajo las mismas condiciones temporales.

  - **Color de las piezas:** Las 100 partidas se jugaron bajo las siguientes condiciones:

    - **Primeras 50 partidas:** Nuestro bot jugó con las blancas y Stockfish con las negras.
    - **Últimas 50 partidas:** Se invirtieron los roles, con nuestro bot jugando con las negras y Stockfish con las blancas.

- **Registro de resultados:** Para cada conjunto de 100 partidas, se registraron los resultados en tres categorías:

  - **Victorias del bot:** El número de partidas ganadas por nuestro bot.
  - **Victorias de Stockfish:** El número de partidas ganadas por Stockfish.
  - **Empates:** El número de partidas que terminaron en empate.

- **Repetición del proceso:** El experimento se repitió para cada nivel de ELO de Stockfish, comenzando desde 1700 y avanzando en incrementos de 100 hasta 2500. Esta repetición permitió una evaluación exhaustiva del rendimiento del bot en diferentes niveles de dificultad.

6.1.3. Análisis de resultados y estimación del ELO del bot

Los resultados obtenidos fueron los siguientes:

ElO de Stockfish	Victorias	Derrotas	Empates
1700	87	13	0
1800	77	22	1
1900	73	23	4
2000	65	34	1
2100	61	35	4
2200	49	47	4
2300	43	51	6
2400	33	60	7
2500	16	72	12

Tabla 6.3: Resultados del bot en diferentes niveles de ElO de Stockfish

Cuya representación de forma gráfica es la siguiente:

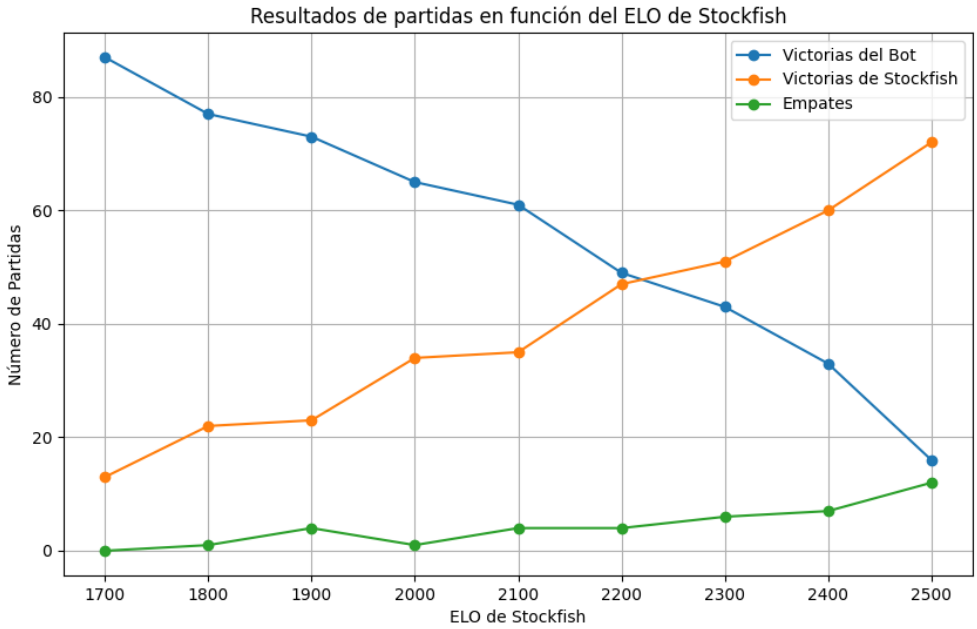

Figura 6.2: Resultados del bot en diferentes niveles de ELO de Stockfish

La gráfica muestra cómo nuestro bot comienza a perder más partidas a medida que el ELO de Stockfish aumenta, observando que a partir de un ELO de 2200, nuestro bot pierde más juegos de los que gana, lo cual sugeriría que el ELO de nuestro bot estaría en torno a los 2200 puntos.

Para conocer de manera más exacta cuál sería el valor del ELO, seguiremos una metodología que estima la probabilidad de victoria en función de la diferencia de ELO entre dos jugadores.

Cálculo de la probabilidad de victoria

La **probabilidad de victoria** de un jugador A frente a un jugador B, según el sistema de ELO, se calcula con la siguiente fórmula:

$$P(A) = \frac{1}{1 + 10^{\frac{(ELO_B - ELO_A)}{400}}}$$

Donde:

- $P(A)$ es la probabilidad de que el jugador A gane.
- ELO_A es el ELO del jugador A.
- ELO_B es el ELO del jugador B.

En nuestro caso, ELO_A corresponde al ELO de nuestro bot, y ELO_B al ELO de Stockfish en las pruebas.

Función de error

Para estimar el ELO de nuestro bot, hemos utilizado una función de error que mide la diferencia entre las probabilidades de victoria observadas y las probabilidades de victoria esperadas según la fórmula de ELO. La **función de error** que utilizamos fue la suma de los errores cuadráticos entre estas probabilidades:

$$\text{Error} = \sum_{i=1}^{n} (\text{Probabilidad Observada}_i - \text{Probabilidad Esperada}_i)^2$$

Donde:

- n es el número de partidas en un conjunto determinado.
- Probabilidad Observada$_i$ es la probabilidad observada de victoria de nuestro bot (calculada como $\frac{\text{Victorias del Bot}}{\text{Total de Partidas}}$).
- Probabilidad Esperada$_i$ es la probabilidad de victoria esperada según la diferencia de ELO entre nuestro bot y Stockfish.

Minimización de la función de error

Para determinar el ELO que mejor se ajusta al rendimiento de nuestro bot, se utilizó una técnica de optimización conocida como **minimización de la función de error**. La idea es encontrar el valor de ELO_{bot} que minimice el error entre las probabilidades observadas y las esperadas.

Al aplicar este método a los datos obtenidos en las pruebas, estimamos que el ELO de nuestro bot se encuentra aproximadamente en **2180** puntos (el valor que más minimiza la función de error), lo que coincide con el análisis gráfico y el rendimiento observado en los diferentes niveles de ELO de Stockfish.

6.2. Conclusión

El desarrollo de un motor de ajedrez basado en inteligencia artificial ha sido un proceso complejo, que ha implicado la implementación de diversos componentes, desde la interfaz de usuario hasta la lógica del juego y, en particular, la inteligencia artificial. A lo largo del proyecto, hemos explorado tanto los aspectos fundamentales de la programación aplicados al ajedrez como los desafíos inherentes a la creación de un bot que fuese desafiante para el jugador.

Uno de los principales logros del proyecto ha sido el desarrollo de un motor de ajedrez que utiliza el algoritmo *minimax* en conjunto con *alpha-beta pruning* y las varias optimizaciones y mejoras que hemos ido implementando para mejorar y optimizar la búsqueda de movimientos, asegurando una toma de decisiones eficiente.

Para estimar el rendimiento de nuestro motor, se realizaron numerosas pruebas aprovechando el motor Stockfish, configurado con distintos niveles de ELO. Estas pruebas nos permitieron no sólo evaluar el rendimiento del motor en términos de partidas ganadas, perdidas y empatadas, sino también realizar una estimación bastante precisa de su ELO. A través de un proceso de ajuste de la probabilidad de victoria basada en el sistema ELO, llegamos a la conclusión de que el ELO de nuestro motor se sitúa alrededor de 2180. Este valor indica que nuestro bot es competitivo contra jugadores avanzados y motores de ajedrez de nivel intermedio, aunque aún se encuentra lejos de los motores de élite, lo cual era de esperar.

Durante el desarrollo del proyecto, enfrentamos varios retos, especialmente en la generación y evaluación de movimientos. Sin embargo, mediante técnicas de prueba, como los *Perft Tests*, logramos verificar que el motor generaba el número correcto de nodos, lo que garantizó su correcto funcionamiento en términos de cálculo de movimientos.

En resumen, este proyecto ha permitido crear una plataforma educativa para el desarrollo de inteligencia artificial aplicada al ajedrez. A pesar de que el rendimiento de nuestro bot no alcanza los niveles de los motores más avanzados, como Stockfish o AlphaZero, el resultado es un motor funcional y competitivo, que proporciona una sólida base para futuros desarrollos. Las posibles mejoras en la evaluación de posiciones, optimización de la búsqueda, o incluso la implementación de redes neuronales para la

evaluación de las posiciones, podrían aumentar significativamente el rendimiento del bot en futuras iteraciones.

En conclusión, este proyecto no sólo ha cumplido los objetivos planteados al principio, sino que también ha proporcionado una valiosa experiencia educativa en el desarrollo de inteligencia artificial aplicada a los juegos de tablero. A través de este trabajo, hemos logrado profundizar en conceptos fundamentales de la programación, la lógica de juegos y las técnicas de búsqueda. Adicionalmente hemos ofrecido una plataforma accesible y bien documentada, que facilita la enseñanza de conceptos complejos de manera intuitiva, promoviendo el desarrollo de habilidades críticas en programación y diseño de juegos. De esta manera, se espera que sirva como una herramienta educativa que inspire a otros a adentrarse en el fascinante mundo de la inteligencia artificial y el desarrollo de videojuegos.

Capítulo 7

Bibliografía

[1] Federación Española de Ajedrez. Leyes del ajedrez y traducciones oficiales. Reglamentación FIDE, 2024. https://feda.org/feda2k16/leyes-del-ajedrez-y-traducciones-oficiales-reglamentacion-fide/.

[2] A. Fox and D.A. Patterson. *Engineering Software as a Service: An Agile Approach Using Cloud Computing*. Strawberry Canyon LLC, 2013.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[4] Stockfish chess engine. https://stockfishchess.org/.

[5] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[6] Alphazero chess engine. https://en.wikipedia.org/wiki/AlphaZero.

[7] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Professional, 3rd edition, 2008.

[8] C# programming language. https://dotnet.microsoft.com/es-es/languages/csharp.

[9] Godot game engine. https://godotengine.org/.

[10] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., USA, 1991.

[11] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[12] Ariel Manzur Juan Linietsky and the Godot community. Node class, 2024. https://docs.godotengine.org/es/4.x/classes/class_node.html.

[13] Ariel Manzur Juan Linietsky and the Godot community. Idle and physics processing, 2024. https://docs.godotengine.org/en/stable/tutorials/scripting/idle_and_physics_processing.html.

[14] Clases estáticas y sus miembros (guía de programación de c-sharp). https://learn.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members.

[15] Chess programming wiki, zobrist hashing. https://www.chessprogramming.org/Zobrist_Hashing.

[16] Godot sprite2d. https://docs.godotengine.org/en/stable/classes/class_sprite2d.html.

[17] Vladimir Fedorovich Demyanov and Vasilii Nikolaevich Malozemov. *Introduction to minimax*. Dover Publications, 1990.

[18] George C. Stockman. A minimax algorithm better than alpha-beta? *Artificial Intelligence*, 12(2):179–196, 1979.

[19] Chess programming wiki negamax. https://www.chessprogramming.org/Minimax.

[20] Chess programming wiki pieces point values. https://www.chessprogramming.org/Point_Value.

[21] Chess programming wiki piece position values (piece-square tables). https://www.chessprogramming.org/Simplified_Evaluation_Function.

[22] Ingo Althöfer. An incremental negamax algorithm. *Artificial intelligence*, 43(1):57–65, 1990.

[23] Chess programming wiki negamax. https://www.chessprogramming.org/Negamax.

[24] Chess programming wiki horizon effect. https://www.chessprogramming.org/Horizon_Effect.

[25] Chess programming wiki quiescence search. https://www.chessprogramming.org/Quiescence_Search.

[26] Chess programming wiki transposition. https://www.chessprogramming.org/Transposition.

[27] Chess programming wiki transposition table. https://www.chessprogramming.org/Transposition_Table.

[28] Wikipedia negamax with transposition table. https://en.wikipedia.org/wiki/Negamax#:~:text=the%20search%20tree.-,Negamax%20with%20alpha%20beta%20pruning%20and%20transposition%20tables,-%5Bedit%5D.

[29] Chess programming wiki late move reductions. https://www.chessprogramming.org/Late_Move_Reductions.

[30] Chess programming wiki late move reductions. https://www.chessprogramming.org/Perft_Results.

[31] Wikipedia elo rating. https://en.wikipedia.org/wiki/Elo_rating_system.

[32] Wikipedia uci protocol. https://en.wikipedia.org/wiki/Universal_Chess_Interface.

[33] Rudolf Huber and Stefan-Meyer Kahlen. Uci protocol specificationsl, 2006. https://backscattering.de/chess/uci/2006-04.txt.