UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERO EN INFORMÁTICA

ENTORNO INTERACTIVO PARA EL ESTUDIO DE ESTRATEGIAS DE I.A. EN JUEGOS

Realizado por

JOSÉ MIGUEL HORCAS AGUILERA

Dirigido por

LAWRENCE MANDOW ANDALUZ

Departamento

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

MÁLAGA, febrero 2012

UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERO EN INFORMÁTICA

Reunido el tribunal examinador en el día de la fecha, constituido por:
Presidente/a Dº/Dª. _____

Secretario/a Dº/Dª. _____

Vocal/a Dº/Dª. _____

para juzgar el proyecto Fin de Carrera titulado: _____

_____

realizado por Dº/Dª. _____

tutorizado por Dº/Dª. _____ ,

y, en su caso, dirigido académicamente por

Dº/Dª. _____

_____

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN

DE _____

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga a _____ de _____ del 20__

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este capítulo describe la motivación del proyecto, así como los objetivos del mismo y la aportación realizada. Se incluye también la estructura de este documento con una breve descripción de los contenidos de cada capítulo.

1.1. Motivación

Los juegos son un tema atractivo a estudiar para los investigadores de Inteligencia Artificial (IA). La naturaleza abstracta de los juegos, la facilidad de representar el estado de los mismos y la definición precisa de sus reglas hace que hayan tenido mucho interés en la comunidad de IA.

Los juegos se pueden representar como problemas de búsqueda con adversarios y son interesantes porque son demasiado difíciles para resolverlos de forma exacta. Normalmente no es factible calcular una solución óptima y requieren la capacidad de tomar alguna decisión.

Los algoritmos y métodos usados en el ámbito de los juegos pueden extenderse a otras áreas de la Inteligencia Artificial o de la Investigación Operativa para problemas de búsquedas o toma de decisiones.

1.2. Estado del arte

Originalmente, los juegos que se han estudiado en IA han sido los clásicos juegos de tablero (*classic board-games*), como el Ajedrez, las Damas, el Othello o el Go. Juegos de dos jugadores, deterministas y de información perfecta. Las estrategias han estado siempre muy ligadas a estas características.

Recientemente, la aparición de nuevas clases de juegos (los llamados *modern board-games* o *Eurogames*, como los "Colonos de Catán" o el "Carcassonne") ha despertado el interés de los investigadores de IA debido a las características de los mismos. Por un lado, en estos juegos las reglas son precisas y se juega por turnos al igual que los juegos clásicos. Por otro lado, pueden incorporar aleatoriedad, ocultación de la información, múltiples jugadores y otras caracterísi-

cas como por ejemplo que el propio tablero se va construyendo mientras se desarrolla la partida o puede variar de una a otra; lo que hace imposible usar libros de aperturas o bases de datos de finales. Estos juegos suponen un enlace directo entre los juegos clásicos y los actuales videojuegos. Los algoritmos clásicos deben modificarse para tener en cuenta estas características.

También han surgido nuevas técnicas, como las estrategias basadas en el método de las simulaciones de Monte-Carlo, que han mejorado los programas de juego, como es el caso del Go donde los programas de ordenador han conseguido llegar al nivel de los humanos en los últimos años. Por ejemplo, en [CBSS08] se aplica el método de Monte-Carlo Tree Search a los clásicos juegos de tablero, a los juegos modernos de tablero y a los videojuegos.

Este proyecto no parte de cero, pues la idea de una aplicación interactiva para estrategias y juegos surge a partir de unos trabajos realizados sobre los contenidos de la asignatura "Inteligencia Artificial e Ingeniería del Conocimiento" de la ETSII de la Universidad de Málaga.[1] En ellos [HA10] se adaptan al lenguaje de programación Java los algoritmos clásicos de juegos desarrollados originalmente en lenguaje Lisp [MA08] junto con su correspondiente documentación en forma de apuntes académicos. Aprovecho para dar las gracias al centro ETSII y a la Universidad de Málaga por la beca, así como a los profesores implicados: Lawrence Mandow (profesor asignado durante las prácticas) y Eva Millán (Subdirectora de Innovación Educativa y coordinadora de los alumnos de prácticas).

1.3. Objetivos

El objetivo del proyecto es desarrollar un entorno interactivo que permita jugar y comparar el rendimiento de estrategias de IA en juegos.

Los juegos a considerar son una clase especializada: juegos de suma cero, de dos jugadores, por turnos, deterministas y de información perfecta. El proyecto incluye el desarrollo de dos de estos juegos:

- El juego del Conecta-4.
- El juego del Go.

La elección de estos juegos se deben a que el Conecta-4 es relativamente sencillo, pues su espacio de estados es pequeño en comparación con otros juegos como el Ajedrez (10^{21} nodos aproximadamente en el árbol de juegos completo del Conecta-4 frente a 10^{123} nodos en el árbol de juegos del Ajedrez). En cambio, el juego del Go supone todo un desafío: el tamaño de su árbol de búsqueda completo es de 10^{360} nodos para un tablero de dimensiones 19×19; aunque en el proyecto se considerará una versión reducida del tablero (9×9), lo que simplifica el espacio de estados en torno a los 10^{80} nodos.

[1] Trabajos realizados por el propio autor del proyecto durante la beca de Prácticas de Gestión para la convergencia de las enseñanzas con el Espacio Europeo de Educación Superior (EEES) durante el periodo comprendido entre los meses de febrero y diciembre del año 2010.

El proyecto se centra en el estudio y desarrollo de las estrategias clásicas como minimax, alfa-beta o las tablas de transposición; y de métodos más recientes como Monte-Carlo Tree Search. Las estrategias desarrolladas son las siguientes:

- Un jugador humano para cada juego.
- Un jugador aletorio.
- Un jugador evaluador heurístico.
- La estrategia minimax.
- La poda alfa-beta.
- Las tablas de transposiciones.
- El método de Monte-Carlo.
- El método de Monte-Carlo Tree Search.

Algunas de estas estrategias incluyen varias versiones con modificaciones, como por ejemplo un límite en el tiempo disponible para decidir el mejor movimiento.

También se incluyen los heurísticos necesarios para las estrategias que lo requieran y que permiten evaluar las posiciones de los juegos. Entre ellos se encuentran un evaluador con tablas de valor y un evaluador con red neuronal que precisan de un entrenamiento previo y por tanto de un módulo de aprendizaje propio. Los demás heurísticos desarrollados son específicos para cada juego pues requieren de información del dominio.

1.4. Aportaciones

El entorno interactivo permite al usuario jugar contra las estrategias desarrolladas o contra otro jugador humano; ver el desarrollo de una partida entre dos estrategias controladas por el ordenador; simular un número de partidas obteniendo estadísticas de los resultados; y analizar detenidamente un único movimiento en un determinado estado de un juego. En definitiva, la aplicación supone una forma más fácil y amena de estudiar, analizar y comparar las estrategias y los propios juegos.

El proyecto proporciona además un marco de trabajo para el dominio de los juegos en IA, permitiendo incorporar otros juegos, estrategias y heurísticos de forma sencilla. Esto hace que el proyecto sea útil tanto para la docencia como para la investigación en el campo de los problemas de búsqueda en juegos.

1.5. Estructura de los contenidos

A continuación se describe brevemente el contenido de cada uno de los capítulos de este documento:

Capítulo 1: Introducción. Se presenta la motivación del proyecto y los objetivos del mismo, así como las aportaciones realizadas; y se realiza un breve repaso a la actualidad en el ámbito de los juegos en IA.

Capítulo 2: Juegos. Se describen las características de los juegos en IA y se representan como problemas de búsqueda con adversarios. Se detallan los dos juegos considerados: el juego del Conecta-4 y el juego del Go.

Capítulo 3: Estrategias. Define el concepto de agente inteligente y estudia en detalle cada uno de los algoritmos y estrategias que han sido desarrollados. Para cada uno de estos algoritmos se definen los agentes jugadores que los utilizan.

Capítulo 4: Heurísticos. Presenta los conceptos de heurístico y evaluador heurístico, definiendo la función de evaluación que necesitan algunas de las estrategias del Capítulo 3. Se estudia el aprendizaje con refuerzo para entrenar los evaluadores con tabla de valor y red neuronal y se proponen varios heurísticos para cada uno de los juegos.

Capítulo 5: Especificación. Este capítulo detalla los requisitos del proyecto y describe los casos de uso de la aplicación interactiva.

Capítulo 6: Arquitectura de la aplicación. Se muestra la arquitectura de la aplicación de forma global, presentando los diagramas de clases más importantes que permiten construir los módulos de razonamiento de los jugadores y el espacio de estados de los juegos.

Capítulo 7: Experimentación. Se muestran los resultados obtenidos en las pruebas realizadas para las diferentes estrategias y juegos: simulaciones de partidas, entrenamientos de jugadores y análisis de posiciones concretas de los juegos.

Capítulo 8: Conclusiones y trabajo futuro. Presenta las conclusiones obtenidas tras el desarrollo del proyecto y describe el trabajo futuro propuesto sobre el proyecto: posibles extensiones y mejoras; además de la aplicación de los contenidos a otros campos de la IA.

Apéndice A: Manual de usuario. El contenido de este apéndice es el manual de usuario de la aplicación interactiva; explica su instalación y su manejo básico.

Apéndice B: Desarrollo de nuevos juegos, estrategias y heurísticos. En este apéndice se explica como desarrollar nuevos juegos, estrategias y heurísticos, de forma que puedan usarse junto con los desarrollados. También se explica la forma de integrar estos elementos en la aplicación interactiva.

Capítulo 2

Juegos

En este capítulo se explican los diferentes tipos de juegos en función de las características que los hacen interesantes para la IA. Nos centraremos en una clase especializada: juegos con búsqueda con adversarios. Concretamente se estudian dos: el juego del Conecta-4 y el juego del Go.

Los juegos proporcionan una tarea estructurada en la que es muy fácil medir el éxito o el fracaso. En comparación con otras aplicaciones de IA, los juegos no necesitan grandes cantidades de conocimiento.

Los juegos son conocidos en IA como **problemas de búsqueda con adversarios** porque se trata de entornos competitivos en los cuales los objetivos de los agentes[1] están en conflicto. Estos problemas se resuelven mediante los denominados "algoritmos de juegos", los cuales se estudiarán en el capítulo 3.

2.1. Características de los juegos

Este proyecto está orientado a una clase de juegos especializada: juegos de suma cero, de dos jugadores, por turnos, deterministas y de información perfecta. Los problemas de juegos en IA pueden clasificarse según estas propiedades:

- **Número de jugadores**
  Un juego puede ser para un jugador sin adversario (conocidos como puzzles o solitarios, por ejemplo el Puzzle-8 o el Cubo de Rubik), para dos jugadores (bipersonales) o para N jugadores.

- **Suma cero**
  Un juego de suma cero describe una situación con adversarios, en la que los valores de utilidad al final del juego, son siempre iguales y opuestos. Esto quiere decir que la

[1] En el capítulo 3 se definirá formalmente el término agente, pero por ahora podemos considerar un agente como un jugador de juegos.

ganancia o pérdida de un agente se equilibra con exactitud con las pérdidas o ganancias de los otros agentes. Por ejemplo, si un jugador gana una partida de Ajedrez (+1), el otro jugador necesariamente pierde (-1).

- **Orden de los movimientos**
  Los agentes pueden realizar sus acciones alternativamente (por turnos), al azar o incluso realizar cada agente un número determinado de acciones seguidas.

- **Información perfecta**
  En un juego con información perfecta no interviene el azar y los estados del juego son totalmente observables, es decir, los agentes tienen un conocimiento perfecto del estado actual del juego en cada momento y es el mismo para todos los agentes. Por el contrario, en un juego con información imperfecta puede intervenir el azar y puede haber ocultación de información entre los agentes.

- **Determinismo**
  Un juego es determinista si no aparece el azar en algunas de las propiedades anteriores. En caso contrario el juego es indeterminista o estocástico.

Teniendo esto en cuenta, los juegos a considerar son aquellos que presenten entornos deterministas, totalmente observables en los cuales hay dos agentes cuyas acciones deben alternar y en los que los valores de utilidad al final son iguales y opuestos. Ejemplos de juegos que cumplen estas propiedades son: el ajedrez, las damas, el 3 en Raya, el Othello o Reversi y por supuesto los dos juegos que se proponen en este proyecto: el Conecta-4 y el Go. En IA, a este tipo de juegos se les conoce como *classic board-games* en la literatura anglosajona.

2.2. Juegos como problemas de búsqueda con adversarios

Un juego puede definirse formalmente como una clase de problemas de búsqueda con los siguientes elementos:

- Un **estado inicial**, que incluye la situación inicial de la partida (por ejemplo la posición inicial del tablero) e identifica al jugador que mueve.

- Una **función sucesor**, que devuelve una lista de pares (movimiento, estado), indicando un movimiento legal y el estado resultante.

- Un **test terminal** que determina cuándo se termina el juego. A los estados donde el juego se ha terminado se les llaman estados terminales.

- Una **función de utilidad** (también llamada función objetivo o función de rentabilidad), que da un valor numérico a los estados terminales. Por ejemplo, +1, -1 ó 0 cuando el resultado de un juego es un triunfo, una pérdida o un empate respectivamente.

Estos elementos permiten construir un **espacio de estados** donde puedan actuar los algoritmos de búsqueda. Los algoritmos interpretan el espacio de estados de los juegos como un **árbol de juegos** o **árbol de búsqueda**. Estos conceptos se definirán en detalle en el capítulo 3.

El tamaño del espacio de estados permite clasificar los juegos por su complejidad. La tabla 2.1 presenta varios juegos clásicos; para cada uno de ellos se muestra la complejidad del árbol de búsqueda en función del número de nodos que contiene y se muestra la clase de complejidad[2] a la que pertenece [Epp].

Tabla 2.1: Complejidad de los juegos clásicos de tablero.

Juego	**Tamaño del árbol de juegos**	**Clase de complejidad**
3 en raya	10^5	PSPACE-completo
Conecta-4	10^{21}	PSPACE
Damas	10^{31}	EXPTIME-completo
Othello o Reversi	10^{58}	PSPACE-completo
Ajedrez	10^{123}	EXPTIME-completo
Go (19x19)	10^{360}	EXPTIME-completo

A continuación se presentan los juegos desarrollados en el proyecto: el Conecta-4 y el Go.

2.3. El juego del Conecta-4

El *Conecta-4* (también conocido como *4 en línea*) es un juego para dos jugadores en el que el objetivo es ser el primero en hacer una línea de cuatro fichas consecutivas del mismo color.

Se ha considerado una versión generalizada del juego con un tablero de n filas $\times$ m columnas y una longitud ganadora de k fichas; aunque las reglas del juego no se ven afectadas por estos parámetros.

2.3.1. Reglas del juego

El juego se desarrolla en un tablero en posición vertical de 6 filas y 7 columnas. Cada jugador dispone de fichas de un color y se turnan para soltarlas en las columnas. Las fichas ocuparan la posiciones más bajas de las columnas. En cada turno sólo puede soltarse una ficha en cualquiera de las columnas siempre que la columna no esté completa.

Un jugador gana cuando consigue colocar cuatro de sus fichas en línea (horizontal, vertical o diagonal). Si todas las columnas se llenan de fichas y ningún jugador ha conseguido conectar cuatro fichas, la partida termina en empate.

[2] *PSPACE* es la clase de problemas resolubles en espacio polinómico. *EXPTIME* es la clase de problemas resolubles en tiempo exponencial. Un problema P es *C-completo* (donde C es la clase de complejidad del problema P) si cualquier otro problema de la clase C es reducible a P.

2.4. El juego del Go

El Go es un juego de mesa estratégico muy popular en Asia. En él, dos jugadores colocan alternativamente fichas negras y blancas (llamadas **piedras**) sobre las intersecciones libres de un tablero de dimensiones 19×19. El objetivo del juego es controlar una porción más grande del tablero que el oponente.

Existen versiones del juego en tableros más pequeños (9×9, 13×13 ó 17×17). El tamaño más común es 19×19 y es el que se usa en los torneos oficiales [IGo82, EUG57]. Aquí se ha considerado una versión generalizada del juego en un tablero de dimensiones $N \times N$.

A continuación se exponen las reglas del juego, basadas en las *"Reglas Tromp/Taylor"* (más conocidas como *"Reglas lógicas del Go"*) que tienen las características de ser muy elegantes y concisas [TT96].

2.4.1. Reglas del juego

Al inicio de la partida el tablero está vacío y se juega con $N \times N$ piedras blancas y $N \times N + 1$ negras, donde $N \times N$ son las dimensiones del tablero (180 piedras blancas y 181 negras para un tablero 19×19). El jugador con las piedras negras juega primero. Después ambos jugadores mueven por turnos. Un movimiento consiste en poner una piedra en una intersección vacía del tablero o bien pasar el turno. Si ambos jugadores pasan el turno consecutivamente, la partida termina.

Dos o más piedras juntas (horizontal o verticalmente) del mismo color forman un **grupo**. Las **libertades** de una piedra o grupo son las intersecciones vacías horizontales o verticales, adyacentes a esa piedra o grupo. Una piedra sola en el medio del tablero tiene cuatro libertades, una piedra en el borde tiene tres libertades y una piedra en la esquina tiene dos.

Una piedra o grupo de piedras se captura y retira del tablero de juego si no tiene libertades, esto es, si se encuentra completamente rodeada de piedras del color contrario. Las piedras capturadas no pueden volver a jugarse durante la partida.

En principio está permitido jugar en cualquier punto del tablero (incluidos los bordes y las esquinas), pero existen dos movimientos prohibidos: el **suicidio** y la **situación de *Ko***.

- **Suicidio**
  Está prohibido ubicar una piedra en una posición que no captura ninguna otra y deja a su propia piedra o grupo sin libertades. En la figura 2.1 se aprecia esta situación, mientras que la figura 2.2 no representa una situación de suicidio.

- **La regla del *Ko***
  La regla del *Ko* evita que las posiciones de las piedras en el tablero se repitan en dos turnos diferentes. Los tableros de la figura 2.3 reflejan esta situación. Si un jugador captura

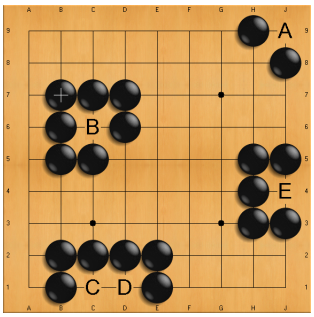


Figura 2.1: Situación de suicidio. Blancas no pueden mover en A, B o E.

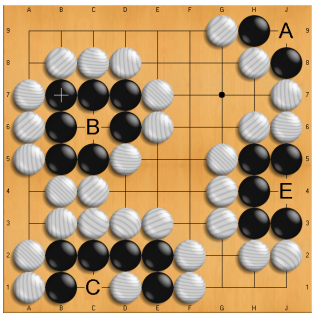
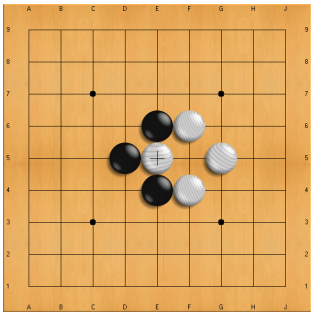


Figura 2.2: Blancas pueden mover en A, B, C o E, pues ha rodeado completamente al enemigo.



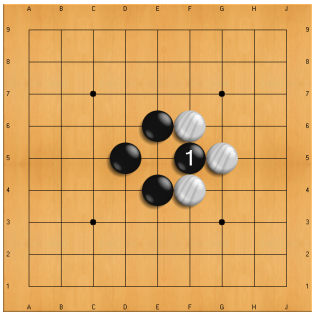


Figura 2.3: Situación de *Ko*.

una piedra en situación de *Ko*, otro jugador no puede recapturar la misma piedra inmediatamente; ha de hacer otra jugada antes de recapturar. Se trata de evitar una situación de infinitud.

Una partida de Go termina cuando ambos jugadores consideran que ya no existen territorios por disputar. Cuando esto ocurre ambos jugadores pasan turno consecutivamente. En una partida profesional, tal circunstancia la decide un juez. Una partida también puede terminar si ambos jugadores han agotado sus piedras.

El **territorio** está formado por las intersecciones del tablero que se encuentren vacías. El **territorio privado** está formado por las intersecciones que se encuentran dentro de algún cerco de uno u otro jugador. El **territorio público** es aquel que no está cercado por ninguno de los jugadores y no influye en la puntuación final. Hay ciertos territorios conquistados que no se encuentran protegidos del todo, pero que si el jugador contrario los atacase perdería de todas formas, luego son parte del territorio privado de quien lo tiene cercado.

El ganador de la partida es el jugador con más puntos.

Puntuación

Existen dos formas de contar los puntos de una partida: según las reglas japonesas y según las reglas chinas:

- **Reglas japonesas**
  Cada jugador recibe un punto por cada intersección vacía dentro de su territorio, menos un punto por cada piedra que haya capturado el enemigo.

- **Reglas chinas**
  Cada jugador recibe un punto por cada intersección vacía dentro de su territorio, más un punto por cada piedra que tenga sobre el tablero.

Ambos métodos producen el mismo resultado. Las reglas de puntuación japonesas son las más extendidas y son las que se usan en los torneos oficiales.

El jugador con piedras negras tiene ventaja debido a que siempre mueve primero. Para compensar esta ventaja se suma una cantidad de puntos al jugador blanco. A estos puntos se les denomina *komi* y son determinados antes de empezar la partida. El valor de *komi* puede variar según los distintos reglamentos; suele oscilar entre 5.5 y 7.5 puntos para un tablero 19x19. Este valor también es usado para evitar que una partida termine en empate.

Handicap

En el Go existe un sistema de *handicap* o ventaja para igualar una partida entre jugadores de diferente nivel.

El *handicap* consiste en un número de piedras colocadas sobre el tablero antes de empezar la partida. En ese caso el jugador de menor nivel jugará con las piedras negras y comenzará con un número determinado de piedras ya colocadas sobre el tablero. Estas piedras se colocan sobre los puntos marcados del tablero, quedando las piedras de forma simétrica. También existen otras variantes de *handicap* libre, en el cual el jugador coloca las piedras de ventaja donde él quiera.

El número de piedras de *handicap* depende de la diferencia de nivel entre los jugadores. Existe una clasificación oficial de niveles para los jugadores de Go [GS11].

Capítulo 3

Estrategias

Este capítulo define el concepto de *agente inteligente* y profundiza en el estudio de los algoritmos que usan los agentes para juegos con adversario. Estos algoritmos representan las estrategias usadas por los agentes para realizar los movimientos válidos en los juegos y lograr sus objetivos.

La siguiente sección comienza definiendo el concepto de agente de forma general.

3.1. Agentes

Un **agente** o **agente inteligente** es un sistema capaz de percibir su entorno, procesar tales percepciones y actuar en su entorno de manera racional.

Un agente tiene por lo tanto un **comportamiento**: se desenvuelve en un **medio**, es capaz de percibir el medio y es capaz de actuar sobre el medio; tiene un **objetivo** y tiene **conocimiento** que le permite tomar una decisión para actuar siguiendo el principio de **racionalidad**: eligiendo siempre la acción que parece acercarle más a su objetivo.

Antes de definir completamente nuestro agente indicando sus objetivos, conocimiento y acciones a realizar, debemos definir el medio en el que se va a desenvolver.

3.2. El medio

El **medio** o **entorno** en el que se desenvuelven nuestros agentes son los juegos con adversario, un subconjunto de los problemas de decisión Markovianos. El entorno está definido por el espacio de estados del problema, en el caso de los juegos podemos considerar el espacio de estados como un árbol de juegos.

Un **árbol de juegos** es una representación de los estados de un juego mediante un *hipergrafo*. Un **hipergrafo** es un tipo de grafo cuyos arcos o aristas (llamadas **hiperarcos** o **hiperaristas**) pueden relacionar a cualquier cantidad de vértices, en lugar de sólo un máximo de dos. Se

caracteriza por la distinción entre 2 tipos de arcos que se denominan arcos *'Y'* y arcos *'O'*. Los arcos *'O'* se corresponden con aquellos definidos y utilizados en los grafos, mientras que un arco *'Y'* puede apuntar a cualquier número de sucesores y representa un conjunto de posibilidades que se deben satisfacer simultáneamente.

En un árbol de juegos los nodos son los estados del juego y las aristas son los movimientos legales. El nodo raíz del árbol es el estado inicial del juego mientras que las hojas del árbol son los estados finales del juego. Un camino en el árbol representa una sucesión de jugadas o movimientos en el juego.

El estado del juego está definido por la "situación del tablero" y el turno (el jugador al que le toca a jugar). Las reglas del juego limitan los movimientos legales del juego. No todos los juegos disponen de un tablero sobre el que realizar movimientos, por ejemplo el juego de *Grundy* [Wei] o los juegos de cartas; por lo que la situación del tablero se refiere a la posición o estado de todos los elementos que intervienen en el juego (fichas, monedas, cartas, tablero, ...) y que permiten diferenciar un estado del juego de otro diferente.

En el capítulo 2 se definió formalmente un juego como una clase de problemas de búsqueda con un estado inicial, una función sucesor, un test terminal y una función de utilidad. El estado inicial y los movimientos legales para cada jugador (función sucesor) definen el árbol de juegos. La figura 3.1 muestra parcialmente el árbol de búsqueda para el juego del *3 en Raya* o *Tic-Tac-Toe*. El nodo raíz es el estado inicial; el primer jugador coloca una *X* en una posición vacía y a continuación se alternan los movimientos con el segundo jugador que coloca *O*; finalmente se alcanzan los estados terminales en los cuales se puede asignar utilidades según las reglas del juego, en este caso +1 si gana el primer jugador (*X*), -1 si gana el segundo jugador (*O*) y 0 si hay empate.

Los juegos a tratar son principalmente juegos bipersonales, de suma cero (si un jugador gana, el otro pierde), de información perfecta y por turnos; tal y como se definieron en el capítulo 2 dedicado a los juegos.

Una vez conocido el medio en el que se desenvuelve el agente podemos completar la definición de un agente inteligente jugador de juegos.

3.3. Agentes para juegos

En los juegos, el agente hará las veces de jugador. Por esa razón, para referirnos a él, se utilizarán indistintamente los términos agente, agente inteligente, jugador o agente jugador. Podemos ahora definir formalmente un agente en el contexto de los juegos.

Un **agente jugador** viene definido por los siguientes elementos:

- **El medio**
  El jugador se desenvuelve en el espacio de estados de un juego (representado mediante el árbol de juegos) donde otro jugador independiente (un adversario) mueve por turnos.

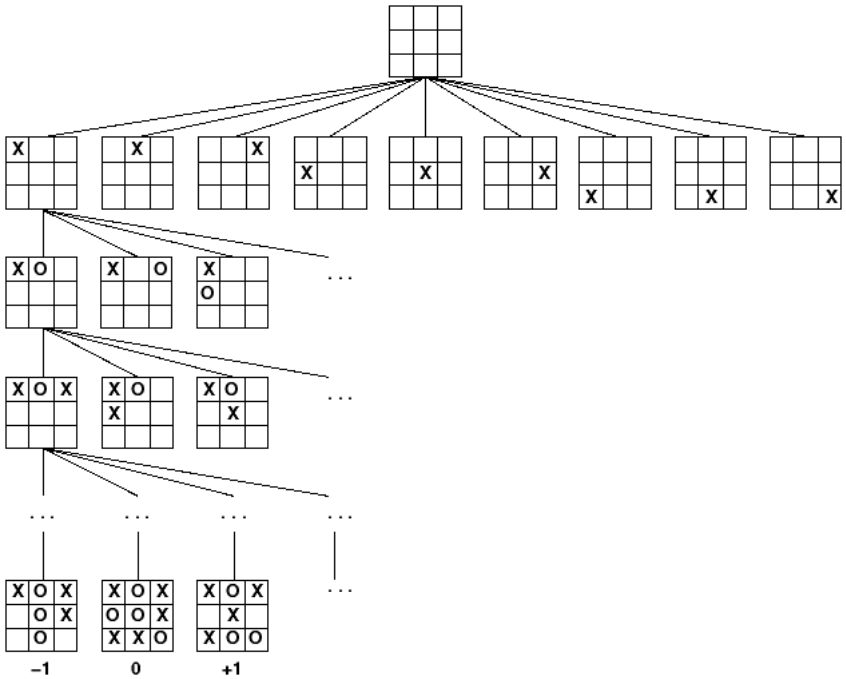


Figura 3.1: Árbol parcial de búsqueda para el *3 en Raya*.

- **Objetivos**
  El objetivo del agente es ganar el juego.

- **Percepciones**
  El agente es capaz de percibir el estado del juego antes de realizar cada movimiento.

- **Acciones**
  El agente es capaz de ejecutar movimientos válidos del juego. Pueden existir limitaciones de tiempo para realizar el siguiente movimiento.

- **Conocimiento**
  El jugador necesita una representación y una *estrategia* que le permitan proponer un movimiento válido conforme a las reglas del juego.

De la sofisticación de la estrategia que use, así como de las limitaciones físicas (por ejemplo, tiempo de cálculo), dependerá la capacidad del agente para alcanzar su objetivo.

3.4. Búsqueda de una estrategia ganadora

Una **estrategia ganadora** es un camino (secuencia de movimientos) desde la raíz del árbol de juegos (estado inicial) hasta las hojas del árbol (estados terminales), que garantiza que el jugador gana siempre. En la terminología introducida en la sección 3.2, una estrategia ganadora es una solución del árbol de juegos que se corresponde con un **hipercamino solución**, es decir, un camino de arcos e hiperarcos tal que todas sus hojas son ganadoras para un jugador.

Encontrar una solución a través de una búsqueda directa desde el estado inicial hasta una posición ganadora no es factible hoy día salvo para juegos muy sencillos (como el *Grundy*), ya que no es posible generar todo el árbol de búsqueda. Incluso para un juego simple como el *3 en Raya* es complejo dibujar el árbol de juegos completo. La tabla 2.1 presentada en el capítulo 2 muestra este hecho.

Por lo tanto no se puede conocer de antemano una estrategia ganadora para un jugador; y en la mayoría de los casos hay que conformarse con obtener una buena jugada a partir de una configuración dada. Esto obliga al agente a tomar algún tipo de decisión para realizar los movimientos del juego.

En la siguiente sección se estudiarán las diferentes estrategias y algoritmos propuestos que usarán los agentes.

3.5. Estrategias

A continuación se estudian los algoritmos y estrategias desarrollados. Todas las estrategias han sido incluidas en la aplicación interactiva para poder usarlas: jugar con ellas, analizarlas y compararlas. Los algoritmos estudiados son versiones generales de los mismos con el objetivo de que sean sencillos de entender y resulten útiles para la docencia. Versiones más sofisticadas de los algoritmos propuestos pueden ser desarrolladas e incluidas en la aplicación como se explica en el apéndice B.

3.5.1. Aleatoria

La estrategia más sencilla es un jugador que realice movimientos aleatorios. Este jugador es totalmente independiente del juego.

Un agente con una estrategia aleatoria obtiene todos los posibles estados sucesores a partir del estado actual del juego y elige uno de ellos de forma aleatoria. Esta forma de proceder es ineficiente para juegos con un factor de ramificación elevado; pero evita que el agente deba conocer cómo se generan los estados para cada juego, ya que los estados sucesores son generados por el propio estado actual del juego, tal y como se definió en 2.2. Esto hace que el agente pueda jugar a cualquier tipo de juego.

Una estrategia aleatoria tampoco ayuda al agente a lograr su objetivo (ganar) de una forma directa; pero será de gran utilidad como base para comparar el rendimiento del resto de estrategias, además de servir como estrategia oponente en los entrenamientos de los agentes que lo requieran.

3.5.2. Evaluador heurístico

Un agente evaluador heurístico es capaz de evaluar heurísticamente si una situación dada le es favorable o no.

El agente emplea la siguiente estrategia de juego: *dada una situación, considera todos los movimientos inmediatos, los evalúa heurísticamente y escoge el mejor.*

Esta estrategia, al contrario que la estrategia aleatoria, no es totalmente independiente del juego, pues necesita de una función de evaluación heurística que depende del tipo de juego.

Cualquier jugador que necesite de una función de evaluación heurística puede considerarse como una especialización de este agente. El capítulo 4 está dedicado a los evaluadores heurísticos que pueden usar los agentes.

3.5.3. Minimax

Antes de presentar los agentes que emplean la estrategia minimax, se detallará el algoritmo minimax de forma general, así como la variante implementada: negamax.

El **algoritmo minimax** proporciona una estrategia óptima, aunque en la práctica no es factible calcularla para juegos complejos. Una **estrategia óptima**, en un problema de búsqueda normal, es una secuencia de movimientos que conducen a un estado objetivo (un estado terminal que es ganador). Sin embargo, en problemas de búsquedas con adversario, el otro jugador también participa y su objetivo es el mismo y opuesto al del primer jugador, por lo que deberá usar una estrategia contingente. Podemos decir que una estrategia óptima conduce a resultados al menos tan buenos como cualquier otra estrategia cuando uno juega con un oponente infalible.

La estrategia minimax consiste en realizar un análisis desde la posición actual y generar una serie de jugadas posibles por parte de ambos jugadores. Una vez alcanzado el nivel de profundidad deseado en el árbol de juegos se utiliza una función de evaluación que asigna valores numéricos a las posiciones finales, lo que permite, además de elegir la mejor posición, transmitir esta información hasta la posición actual que corresponde a la raíz del árbol generado. Minimax sólo tiene en cuenta una visión local del árbol de búsqueda.

Llamaremos al primer jugador *MAX* y al segundo jugador *MIN*; y etiquetaremos los niveles del árbol con *MAX* y *MIN* según le toque jugar a *MAX* o *MIN* respectivamente.

Considerando un árbol de juegos, la estrategia óptima puede determinarse examinando el valor minimax de cada nodo, que llamamos *VALOR_MINIMAX(n)*. El valor minimax de un nodo es la utilidad (para *MAX*) de estar en el estado correspondiente, asumiendo que ambos jugadores juegan óptimamente desde allí hasta el final del juego. El valor minimax de un estado terminal es su utilidad. Considerando una opción, *MAX* prefiere mover a un estado de valor máximo, mientras que *MIN* prefiere un estado de valor mínimo. Se define entonces el *VALOR_MINIMAX(n)* de un nodo como:

$$VALOR\_MINIMAX(n) = \begin{cases} UTILIDAD(n) & \text{si } n \text{ es un estado terminal} \\ \max_{s \in Sucesores(n)} VALOR\_MINIMAX(s) & \text{si } n \text{ es un estado } MAX \\ \min_{s \in Sucesores(n)} VALOR\_MINIMAX(s) & \text{si } n \text{ es un estado } MIN \end{cases} \quad (3.1)$$

La figura 3.2 muestra el funcionamiento del algoritmo minimax para un árbol de juegos. Los nodos □ son «nodos *MAX*», en los que le toca mover a *MAX*, y los nodos ◯ son «nodos *MIN*». Los nodos terminales muestran los valores de utilidad para *MAX*; los otros nodos son etiquetados por sus valores minimax.

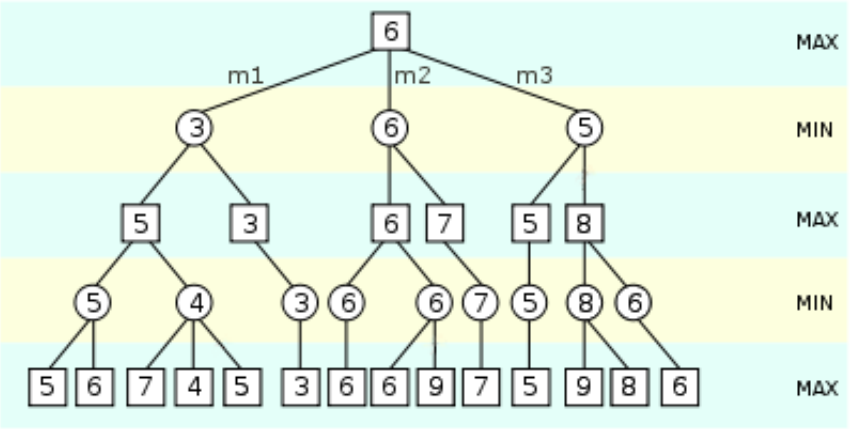


Figura 3.2: Minimax aplicado a un árbol de juegos.

Se realiza una exploración hacia delante, generando primero el árbol completo a una determinada profundidad (cuatro en este caso), a continuación se evalúan las hojas con la función de utilidad y los valores se propagan hacia arriba de la siguiente forma:

- Un nodo *MAX* toma como valor el máximo de sus hijos.

- Un nodo *MIN* toma como valor el mínimo de sus hijos.

Finalmente, el movimiento de *MAX* es el mejor valorado de sus hijos inmediatos, que en el caso de la figura 3.2 se trata del movimiento *m2*.

Esta forma de proceder es más fiable que evaluar solamente los hijos inmediatos, como hacía el agente evaluador heurístico presentado en la sección 3.5.2. La función de utilidad que usa minimax para asignar los valores a los nodos evaluados es una función de evaluación heurística, puesto que en la mayoría de los casos los estados a evaluar no serán terminales. Esta función se define en el capítulo 4 dedicado a los evaluadores heurísticos.

El algoritmo minimax realiza una exploración primero en profundidad. Para una búsqueda en un árbol a profundidad p y un factor de ramificación b, minimax evaluará b^p nodos hojas. La complejidad en tiempo del algoritmo minimax es $O(b^p)$; y la complejidad en espacio es $O(bp)$ si se generan todos los sucesores a la vez, o $O(p)$ si se generan uno por uno. Para juegos reales, los costes de tiempo son impracticables, pero este algoritmo sirve como base para el análisis matemático de juegos y para otros algoritmos más prácticos.

Existen numerosas mejoras del algoritmo minimax original; algunas de ellas consisten en realizar:

- una búsqueda en profundidad con retroceso (*backtrack*),
- una búsqueda con profundización progresiva (*algoritmos anytime*),
- explorar sólo la parte imprescindible del árbol (*poda alfa-beta*), o
- almacenar información sobre los estados visitados anteriormente y usarla cuando nos encontremos esos estados por otro camino del árbol (*tablas de transposiciones*).

A continuación se detallan cada una de estas mejoras que se han implementado e incorporado a la aplicación; presentando primero la variante implementada del algoritmo minimax: negamax.

Negamax

En lugar del algoritmo minimax propiamente dicho se ha implementado una variante equivalente conocida como **negamax**. En esta variante, cada nodo del árbol de juegos toma siempre el valor máximo de sus hijos, independientemente de que sea un nodo *MAX* o *MIN*. Adicionalmente, los valores de los nodos *MAX* se cambian de signo. De este modo, es equivalente para un nodo *MIN* tomar el máximo de las evaluaciones cambiadas de signo, que tomar el mínimo de las evaluaciones sin alterar. El origen de negamax se basa en la igualdad matemática:

$$\text{máx}\,(x,y) = -\,\text{mín}\,(-x,-y)$$

Negamax no supone una mejora directa de minimax, sino que se trata más bien del algoritmo minimax comprimido que evita tener que definir dos funciones distintas para *MAX* y *MIN*. Además, se sigue teniendo el mismo problema de la complejidad que supone explorar el árbol de juegos al completo. Para evitar esto se ha modificado la versión original de negamax dando lugar a dos agentes que emplean dos estrategias diferentes: una con profundidad máxima de búsqueda y otra con un límite de tiempo.

Minimax con profundidad máxima de búsqueda

El primer agente emplea una estrategia con la variante negamax y un parámetro adicional que indica la profundidad máxima de búsqueda en el árbol de juegos a partir de la posición actual.

La estrategia comienza en la posición actual y realiza una búsqueda en profundidad con retroceso (*backtrack*) hasta la profundidad de búsqueda indicada. Los sucesores son generados aleatoriamente de uno en uno y en cada paso el algoritmo se queda con el mejor movimiento en función del valor minimax devuelto.

Este agente puede evaluar posiciones a una profundidad dada. Sin embargo, incrementos lineales de profundidad pueden originar búsquedas que requieran un tiempo exponencialmente mayor. Del mismo modo, aún sin variar la profundidad, el tiempo necesario para la búsqueda

puede ser muy variable. Normalmente el número de posibilidades al comienzo de un juego es mucho mayor que a medida que se desarrolla la partida, por lo que examinar los movimientos iniciales puede llevar un tiempo mayor.

La limitación del tiempo de cómputo puede superarse empleando un algoritmo que en cualquier momento pueda dar una solución, pero que proporcione soluciones mejores cuanto mayor sea el tiempo disponible. Los algoritmos que presentan esta propiedad se conocen como algoritmos *anytime*. El siguiente agente, que también emplea una estrategia minimax, dispone de un límite de tiempo para realizar la búsqueda, en lugar de un límite en la profundidad máxima de búsqueda.

Minimax con límite de tiempo

El segundo agente también emplea negamax pero dispone de un tiempo limitado para realizar la búsqueda del mejor movimiento en el árbol de juegos.

Se trata de un algoritmo *anytime*. Un **algoritmo *anytime*** devuelve una solución cuya calidad depende del tiempo de cómputo disponible. El algoritmo encontrará mejores soluciones a medida que aumenta el tiempo disponible para realizar la búsqueda.

La estrategia realiza una búsqueda con retroceso y profundización progresiva (*Backtrack with Iterative Deepening*. La búsqueda dispone de una cota (profundidad máxima de búsqueda) que se incrementa automáticamente en cada iteración. El algoritmo guarda siempre la mejor jugada inmediata para la última exploración realizada con objeto de devolverla cuando el tiempo se acabe. En cada iteración el algoritmo comprueba el tiempo restante, terminando la búsqueda cuando este se agota.

Puede darse el caso de que el límite de tiempo sea tan pequeño que la estrategia no tenga tiempo de devolver un movimiento, en ese caso el agente acabará realizando un movimiento aleatorio. Además, para asegurarse de devolver el mejor movimiento, la estrategia debe evaluar todos los nodos del nivel del árbol que corresponde a la cota de la iteración actual. Si en la iteración actual no ha tenido tiempo de evaluarlos todos, devolverá el mejor movimiento obtenido en la iteración anterior.

Por simplicidad, la unidad de tiempo escogida ha sido el segundo, pues para la mayoría de juegos no tiene sentido dejar menos tiempo de cómputo al agente debido a la complejidad de los propios juegos. Esto implica que el tiempo mínimo permitido sea de un segundo.

Una vez vista la estrategia básica minimax y su variante negamax, dedicaremos la siguiente sección a estudiar otra mejora de minimax: la poda alfa-beta.

3.5.4. Poda alfa-beta

Esta sección estudia la técnica de la poda alfa-beta como una mejora del algoritmo minimax. También define los agentes que emplearán esta estrategia, que al igual que ocurría con minimax

serán dos: uno con límite de profundidad en la búsqueda y otro con un límite en el tiempo de cómputo.

La **poda alfa-beta** calcula el mismo movimiento que devolvería minimax sin necesidad de examinar todos los nodos en el árbol de juegos, es decir, poda las ramas del árbol que no influyen en la decisión final.

Esta estrategia dispone de dos parámetros (α y β) que describen los límites sobre los valores que se propagan hacia arriba en el árbol:

- α es el valor de la mejor opción (es decir, el valor más alto) que se ha encontrado hasta ahora en cualquiera de los estados elegidos para *MAX*.
- β es el valor de la mejor opción (es decir, el valor más bajo) que se ha encontrado hasta ahora en cualquiera de los estados elegidos para *MIN*.

La poda alfa-beta realiza una búsqueda primero en profundidad, con retroceso y de izquierda a derecha. Para cada nodo se considera un intervalo de posibles valores [α, β]. La búsqueda actualiza estos valores según recorre el árbol: actualiza el valor α (en los nodos *MAX*) y el valor β (en los nodos *MIN*). Cuando encuentra un nodo con un valor peor que su antecesor, la búsqueda se detiene y esa rama del árbol no se examina. Así, se puede podar:

- Debajo de un nodo *MIN* cuando un nodo *MAX* antecesor suyo tenga un valor $\alpha \geq \beta$; lo que se conoce como **corte** α.
- Debajo de un nodo *MAX* cuando un nodo *MIN* antecesor suyo tenga un valor $\beta \leq \alpha$; lo que se conoce como **corte** β.

La figura 3.3 muestra el funcionamiento de la poda alfa-beta para el mismo árbol de juegos de la figura 3.2. Los nodos *MAX* muestran el valor final de α mientras que los nodos *MIN* muestran el valor final de β. El valor minimax devuelto por la poda alfa-beta para el nodo raíz es el mismo que el valor devuelto por el algoritmo minimax y en este caso, el movimiento es también el mismo (*m2*).

En los agentes desarrollados esto no siempre es así y puede ocurrir que el mejor movimiento devuelto por minimax sea distinto del mejor movimiento devuelto por alfa-beta (pero ambos tendrán el mismo valor de evaluación); esto se debe a que los sucesores son generados en orden aleatorio y además, en el caso de que más de un estado tenga la misma evaluación, minimax cogerá el último sucesor que evalúe mientras que alfa-beta puede podar esa rama si encontró otro nodo con la misma evaluación.

La eficacia de la poda alfa-beta depende mucho del orden en el que se examinan los sucesores. Una posible mejora del algoritmo sería examinar primero los sucesores que probablemente sean mejores. Para una búsqueda en un árbol a profundidad p y un factor de ramificación b, en el mejor caso (los nodos están ordenados de mejor a peor), alfa-beta tiene que examinar $O(b^{p/2})$

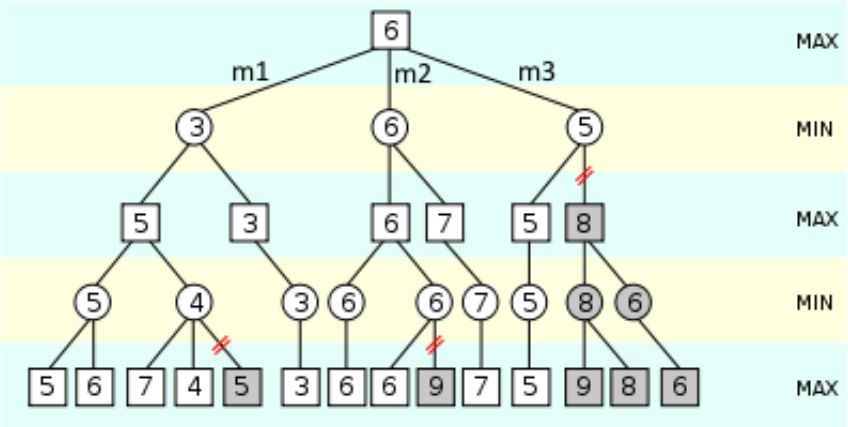


Figura 3.3: Poda alfa-beta aplicada a un árbol de juegos.

nodos para escoger el mejor movimiento, en vez de $O(b^p)$ para minimax.En los algoritmos desarrollados donde los sucesores se examinan en orden aleatorio, el número total de nodos examinados es de $O(b^{3p/4})$. Por último, en el peor de los casos, donde no se realiza ninguna poda, la complejidad es la misma que en minimax: $O(b^p)$.

A continuación se presentan los agentes que emplean la técnica de la poda alfa-beta, al igual que ocurría con minimax se trata de dos agentes con características diferentes: uno con un límite en la profundidad de búsqueda y otro con un tiempo limitado de búsqueda.

Alfa-beta con profundidad máxima de búsqueda

Este agente emplea el algoritmo negamax con poda alfa-beta incluida para realizar una búsqueda en el árbol de juegos a una profundidad máxima indicada a partir de la posición actual.

La lógica del agente es la misma que la explicada para el agente minimax con profundidad máxima de búsqueda (apartado 3.5.3), con el incentivo de que incorpora la poda alfa-beta.

Alfa-beta con límite de tiempo

El segundo agente también emplea el algoritmo negamax con poda alfa-beta y dispone de un tiempo limitado para realizar la búsqueda del mejor movimiento en el árbol de juegos.

El agente tiene las mismas características que el agente minimax con límite de tiempo (descrito en el apartado 3.5.3), pero incluyendo además las ventajas de la poda alfa-beta.

Otra mejora con respecto al algoritmo minimax original y de alfa-beta es la posibilidad de incorporar una tabla de transposición a la búsqueda. En la siguiente sección se estudian las tablas de transposición.

3.5.5. Tablas de transposición

Esta sección desarrolla los conceptos de *transposición* y *tabla de transposición* para las búsquedas en árboles de juegos. Se presentan también los agentes que emplean esta técnica.

Una **transposición** es una permutación diferente de una secuencia de movimientos que termina en la misma posición. En el árbol de juegos se trata de un estado que puede ser alcanzado por más de un camino distinto.

Los estados repetidos en el árbol de búsqueda pueden causar un aumento exponencial del coste de la búsqueda. Es por ello que merece la pena almacenar la evaluación de ese estado en una tabla la primera vez que se encuentre, de modo que no tenga que volver a calcularse la próxima vez que se visite. A esta tabla se le conoce como tabla de transposición.

Una **tabla de transposición** o **tabla de transposiciones** es una base de datos donde se almacenan los resultados de búsquedas previamente realizadas. Normalmente se implementa mediante una tabla *hash* de gran capacidad y se almacenan los estados previamente evaluados, hasta qué nivel se les realizó la búsqueda y qué acción se determinó para estos; aunque pueden almacenar más información si es necesario.

Se trata de una forma de reducir el espacio de búsqueda. Cuando aparece una transposición, se busca en la tabla qué se determinó la última vez; evitando repetir de nuevo toda la búsqueda y pudiendo invertir ese ahorro de tiempo en aumentar la profundidad de búsqueda.

La utilización de una tabla de transposición puede tener un efecto espectacular en situaciones donde hay muchas posibles transposiciones como en la etapa final de los juegos (por ejemplo en el Ajedrez). Por otra parte, el único problema de las tablas de transposiciones es su consumo en memoria. Para que sean realmente útiles deben contener muchas posiciones y si se evalúan un millón de nodos por segundo no es práctico almacenar todos ellos en la tabla de transposición.

Los agentes que emplean las tablas de transposición son los mismos que se han presentado anteriormente para minimax y alfa-beta pero incorporando esta nueva característica, lo que da lugar a cuatro nuevos agentes que se presentan brevemente en los siguientes apartados.

Minimax con profundidad máxima de búsqueda y tabla de transposición

Se trata de un agente que realiza una búsqueda en el árbol de juegos mediante el algoritmo minimax, dispone de un límite en la profundidad máxima de búsqueda a partir del estado actual y cuenta con una tabla de transposición para almacenar los estados evaluados y usar la información guardada en caso de encontrar una transposición.

Para más información sobre este agente se puede consultar el agente minimax con profundidad máxima de búsqueda en la sección 3.5.3.

Minimax con límite de tiempo y tabla de transposición

Este agente realiza una búsqueda en el árbol de juegos empleando el algoritmo minimax, dispone de un límite de tiempo para devolver el mejor movimiento y cuenta con una tabla de transposición.

La sección 3.5.3 contiene información sobre el agente minimax con límite de tiempo.

Alfa-beta con profundidad máxima de búsqueda y tabla de transposición

Este agente emplea el algoritmo minimax con poda alfa-beta incluida para realizar la búsqueda en el árbol de juegos; tiene una profundidad máxima de búsqueda y se ayuda de una tabla de transposición para evitar evaluar los estados repetidos.

En la sección 3.5.4 se detalla el agente alfa-beta con profundidad máxima de búsqueda, que cuenta con las mismas características que este, salvo obviamente, la tabla de transposición.

Alfa-Beta con límite de tiempo y tabla de transposición

El último agente también usa el algoritmo minimax con poda alfa-beta; tiene un límite de tiempo para realizar la búsqueda en el árbol y también cuenta con una tabla de transposición.

El agente alfa-beta con límite de tiempo puede consultarse en la sección 3.5.4.

A continuación se describen los agentes que no necesitan de un evaluador heurístico para determinar el mejor movimiento, estos son los agentes que usan el método de Monte-Carlo y Monte-Carlo Tree Search. Ambas estrategias son totalmente independientes del tipo de juego, ya que no necesitan de una función de evaluación heurística.

3.5.6. Monte-Carlo

Esta sección explica el método de Monte-Carlo aplicado a los problemas de búsqueda en árboles de juegos. Se estudia una versión básica del mismo y se presentan los agentes que lo usan.

El **método de Monte-Carlo** consiste en realizar un número de simulaciones a partir del estado actual para decidir el próximo movimiento.

Una **simulación** es una partida al azar completa del juego desde la posición actual; es decir, partiendo del estado actual en el árbol de juegos, se realizan movimientos aleatorios hasta llegar a un estado terminal, donde se asigna el valor de utilidad (+1, -1 ó 0 si es el estado es ganador, perdedor o empate para el jugador). Este valor es usado como la recompensa esperada que se puede obtener a partir del estado sucesor elegido. Para estimar el valor final de un estado se toman estadísticas haciendo un promedio de las recompensas obtenidas en las simulaciones.

El método genera una lista de posibles movimientos (sucesores del estado actual) y para cada movimiento realiza miles de simulaciones (partidas al azar), obteniendo el valor de recom-

pensa. El movimiento con un valor de recompensa mayor es el elegido como mejor movimiento, es decir, aquel que conduce a la mejor serie de partidas al azar para el jugador actual.

La ventaja de esta técnica es que requiere poco conocimiento del entorno, pero se incrementan los requisitos computacionales (procesador y memoria). Para que el método sea efectivo se deben realizar muchísimas simulaciones, ya que los movimientos de las simulaciones se generan al azar y es posible que una buena jugada sea evaluada erróneamente como una mala jugada.

A priori es difícil establecer de antemano un número determinado de simulaciones, por lo que se suele dejar un tiempo máximo para realizar las simulaciones.

Los siguientes apartados presentan dos agentes que usan el método de Monte-Carlo: uno con un número determinado de simulaciones y otro con un límite de tiempo para realizar las simulaciones.

Monte-Carlo con número de simulaciones fijas

El jugador Monte-Carlo más simple es aquel que tiene un número de simulaciones determinado.

Partiendo del estado actual juega el número de partidas indicadas para evaluar el mejor movimiento. Para cada sucesor del estado actual se realiza un número diferente de simulaciones, pues estos también son elegidos aleatoriamente. El número de simulaciones debe ser suficientemente grande para asegurarse de que se realizan aproximadamente el mismo número de simulaciones para cada posible movimiento. Si N es el número total de simulaciones a realizar y p es el número de sucesores del estado actual, se espera que se realicen p/N simulaciones para cada movimiento. Esto sólo se cumple cuando N tiende a infinito.

Asignar de antemano un número determinado de simulaciones es una tarea complicada. Juegos con un espacio de estados mayor requieren mayor número de simulaciones, pero el tiempo de cómputo necesario también es mucho mayor.

El siguiente agente incorpora un límite de tiempo para realizar las simulaciones.

Monte-Carlo con límite de tiempo

El segundo agente que emplea el método de Monte-Carlo dispone de un límite de tiempo para realizar simulaciones.

Se trata de otro algoritmo *anytime*; devuelve mejores soluciones a medida que aumenta el tiempo disponible para realizar la búsqueda, en este caso para realizar las simulaciones.

Asignar un tiempo límite para Monte-Carlo tampoco es tarea sencilla pues no conocemos el tiempo necesario que necesita cada simulación. La simulaciones deben completarse totalmente para que sean válidas. No tiene sentido dejar un sólo segundo para realizar simulaciones si cada

simulación dura más de un segundo. En ese caso el tiempo disponible del agente se extenderá hasta el tiempo necesario para terminar la simulación.

Por simplicidad, al igual que ocurre con la estrategia minimax con tiempo limitado (seccion 3.5.3), la unidad de tiempo escogida ha sido el segundo.

Una vez presentado el método básico de Monte-Carlo, se explica una versión mejorada del mismo: Monte-Carlo Tree Searh. También se describen los agentes que usan esta nueva versión.

3.5.7. Monte-Carlo Tree Search

Esta sección detalla la estrategia Monte-Carlo Tree Search, basada en el método tradicional de Monte-Carlo explicado en la sección anterior. Se describen también los dos agentes desarrollados que usan esta técnica.

Monte-Carlo Tree Search (MCTS a partir de ahora) consiste en realizar un número de simulaciones a partir del estado actual para decidir el próximo movimiento, igual que hace el método básico de Monte-Carlo; pero dispone además de un árbol propio para almacenar la información obtenida en las simulaciones y poder usarla para mejorar las propias simulaciones y por tanto la decisión final.

Para no confundir el árbol de búsqueda de los propios juegos con el árbol de búsqueda que emplea el método MCTS llamaremos a este último: árbol de Monte-Carlo (árbol MC).

Antes de cada simulación, MCTS realiza primero una búsqueda en el árbol de juegos a partir de la posición actual hasta encontrar un nodo que no se encuentre en el árbol MC (inicialmente el árbol MC está vacío). Para realizar esta búsqueda se utiliza una estrategia o política basada en la información almacenada en el árbol MC. El nuevo estado encontrado se añade al árbol MC y a partir de esa posición se realiza una simulación completa hasta el final de la partida, obteniendo el valor de utilidad del estado terminal. Con el valor de utilidad obtenido se actualiza la información de los estados visitados en el árbol MC durante la simulación. La figura 3.4

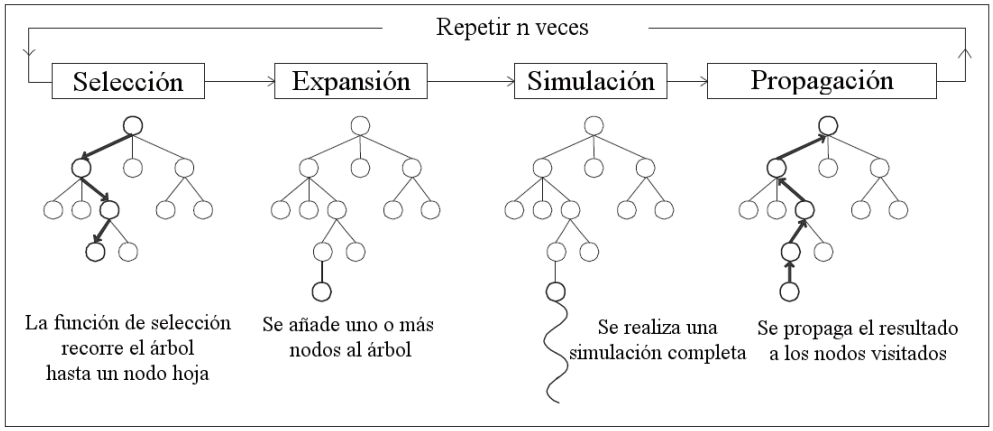


Figura 3.4: Fases del algoritmo Monte-Carlo Tree Search. (Imagen adaptada de [CBSS08].)

muestra este proceso, que se puede dividir en cuatro fases: selección, expansión, simulación y propagación. Antes de detallar cada una de estas fases conviene definir la información que almacenará el árbol MC en cada nodo.

Sea s el estado actual, el nodo correspondiente en el árbol MC contiene tres valores:

- Número total de simulaciones realizadas desde el estado s, que denotamos como *N(s)*.
- Número total de simulaciones en las que se selecciona el movimiento a desde el estado s, que llamamos *N(s,a)*.
- El valor Monte-Carlo del estado (valor MC), representado por *Q(s,a)*, es el resultado promedio de todas las simulaciones realizadas en las que se seleccionó el movimiento a en el estado s:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a) z_i$$

  donde z_i es el valor de utilidad devuelto por la i-ésima simulación, y $\mathbb{I}_i(s,a)$ es un indicador que vale 1 si el movimiento a fue seleccionado en el estado s durante la simulación i y 0 en otro caso. Nótese que $N(s,a) = \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a)$.

La información almacenada puede variar dependiendo de la estrategia de selección que se use en la primera fase, aunque aquí se ha usado la información genérica que se puede obtener mediante las simulaciones, es decir, sin emplear conocimiento experto sobre el entorno.

A continuación se detalla cada una de las fases del algoritmo MCTS:

1. **Selección**

   Partiendo del estado actual, se realiza una búsqueda en el árbol de juegos hasta encontrar un nodo que no está en el árbol MC. Los movimientos realizados en esta búsqueda están elegidos acorde a la información almacenada en el propio árbol MC, siguiendo una determinada estrategia.

   La estrategia elegida debe mantener un equilibrio entre explotación y exploración. Se puede seleccionar en cada paso el mejor movimiento, lo que favorecerá la explotación; pero por otro lado, los movimientos menos prometedores todavía tienen que ser explorados a mayor profundidad, debido a la incertidumbre de la evaluación, lo que favorecería la exploración.

   En función de la estrategia escogida, obtendremos diferentes versiones del algoritmo MCTS. Al final de la sección se citan algunas extensiones de MCTS.

   Usaremos una estrategia conocida como *optimism in the face of uncertainty*[1], que favorece los movimientos con un valor más alto pero permite a su vez explorar aquellas acciones que todavía no han sido suficientemente exploradas. Para ello, definimos $Q^{\oplus}(s,a)$ como

[1] Puede traducirse como *optimismo frente a la incertidumbre*.

el valor del movimiento a desde el estado actual s (valor MC) con una bonificación adicional que será más alta para los movimientos menos visitados:

$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}}$$

donde c es una constante de exploración con valores reales en el intervalo [0,1] y *log* es el logaritmo natural (base e).

El movimiento elegido, que llamamos a^*, será aquel que maximize el valor $Q^{\oplus}(s,a)$:

$$a^* = \max_a Q^{\oplus}(s,a)$$

2. **Expansión**
   Cuando se encuentra un estado que no aparece en el árbol MC, este se añade como un nuevo nodo. De esta forma el árbol se expande un nodo en cada simulación.

   Una variante puede ser añadir al árbol MC cada estado que se visite en la búsqueda, aunque en la práctica, para reducir los requisitos de memoria, no se añaden todos los nodos en cada simulación. Normalmente, sólo se añade un nodo al árbol MC en cada simulación: el primer nodo encontrado que no esté presente en el árbol MC. Si aún así la limitación de la memoria supone un problema, es posible realizar varias simulaciones antes de añadir un nuevo nodo al árbol MC; o incluso podar antiguos nodos del árbol MC a medida que la búsqueda progresa.

3. **Simulación**
   A partir del nuevo estado almacenado en el árbol MC se realiza una simulación completa de la partida siguiendo la estrategia por defecto, esto es, realizando movimientos aleatorios hasta el final del juego. Después se asigna el valor de utilidad z al estado terminal: +1 si es ganador para el jugador, -1 si es perdedor y 0 si es empate.

   Obviamente, también puede emplearse en esta fase otra estrategia distinta para elegir los posibles movimientos, como una función heurística que asigne valores más altos a aquellos movimientos que tienen más probabilidad de ser jugados o que parecen más prometedores; pero esta mejora se sale de los objetivos del proyecto por lo que los agentes desarrollados usan la estrategia por defecto para esta fase.

4. **Propagación**
   Una vez realizada la simulación, se actualiza la información de cada nodo en el árbol MC

en función del valor de utilidad obtenido:

$$N(s) \leftarrow N(s) + 1$$
$$N(s,a) \leftarrow N(s,a) + 1$$
$$Q(s,a) \leftarrow Q(s,a) + \frac{z - Q(s,a)}{N(s,a)}$$

El movimiento final seleccionado por MCTS en el estado actual será el movimiento mejor valorado en el árbol MC, es decir, el movimiento a con un valor MC mayor.

La figura 3.5 muestra cinco simulaciones de MCTS en las que se va construyendo el árbol MC. Se trata de una versión simplificada del algoritmo. Siguiendo con la misma terminología usada en minimax, los nodos de color negro corresponden a nodos *MAX*, los blancos a *MIN*. Cada simulación obtiene un valor de utilidad (en el caso de la figura 3.5 el valor es 1 si gana *MAX* y 0 si gana *MIN*). En cada simulación se añade un nuevo nodo al árbol y se actualiza el valor de cada nodo en el árbol con el valor de utilidad; también se actualiza el número de veces que se ha seleccionado cada nodo. A medida que el árbol MC crece con cada simulación, los valores de los nodos se aproximan al valor minimax real y por tanto la estrategia basada en las simulaciones también se aproxima a la estrategia óptima de minimax.

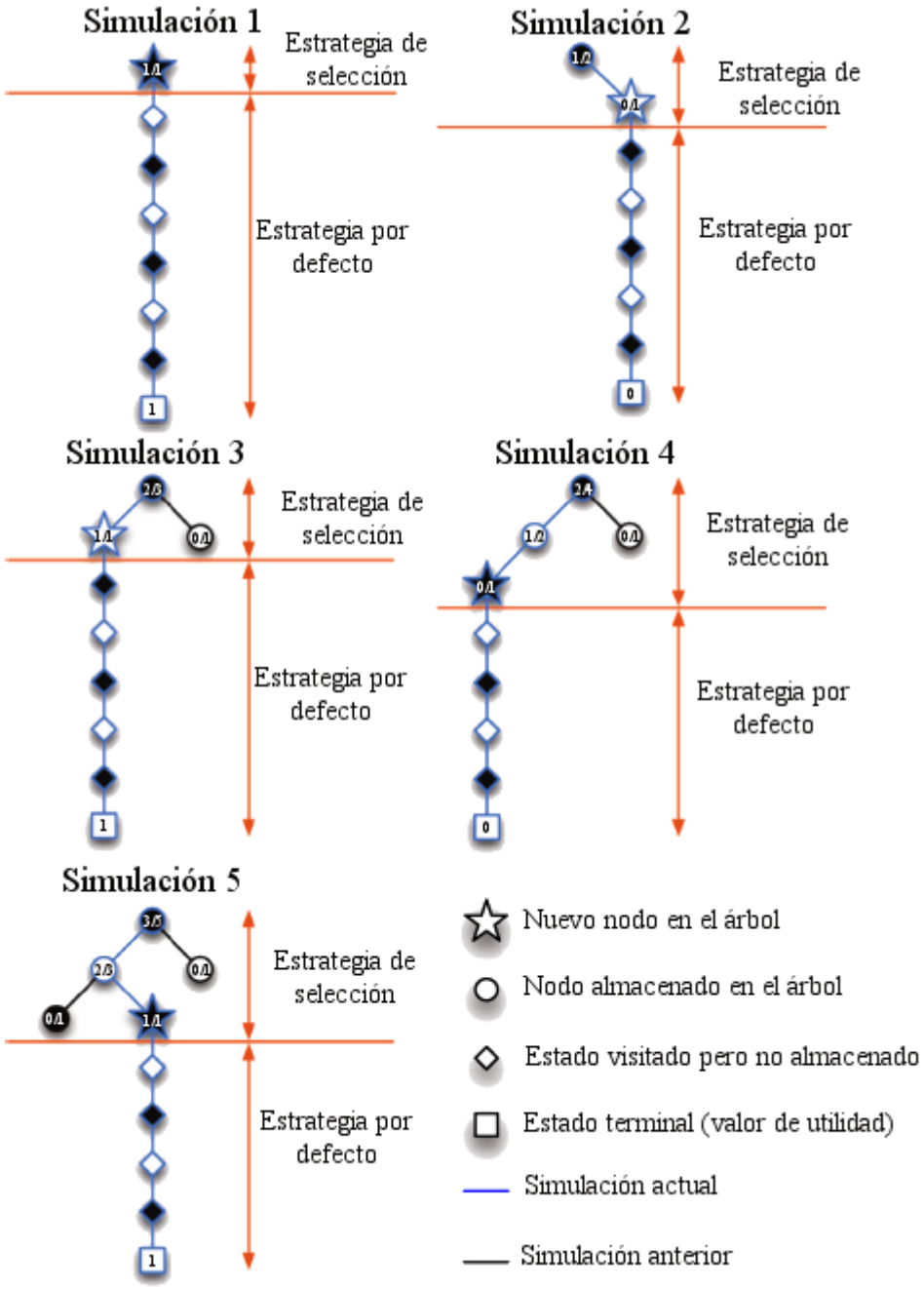


Figura 3.5: Cinco simulaciones de Monte-Carlo Tree Search. (Imagen adaptada de [GS11].)

A continuación se presentan brevemente cuatro extensiones de MCTS en función de la estrategia elegida en las fases de selección y simulación.

Extensiones de Monte-Carlo Tree Search

El rendimiento de MCTS puede mejorarse significativamente si se incorpora información del dominio a la estrategia usada en la fase de selección o incluso a la estrategia por defecto usada en la fase de simulación. Existen varias extensiones de MCTS basadas en la estrategia elegida para ambas fases, algunas extensiones son:

- ***Greedy MCTS***
  La versión más básica de MCTS usa una estrategia *best-first* (primero el mejor) que favorece la explotación frente a la exploración en la fase de selección. A este algoritmo se le conoce como *greedy MCTS*.

- **Algoritmo *UCT***
  El algoritmo *UCT* (*Upper Confidence bounds applied to Trees*, traducido como *límites superiores de confianza aplicados a los árboles*). Usa el principio *optimism in the face of uncertainty* que puede usarse para explorar el árbol de forma eficiente, ya que favorece a los movimientos con mayor valor potencial. Nuestra versión MCTS desarrollada se basa en este algoritmo.

- ***Heuristic MCTS***
  Utiliza funciones heurísticas con información del dominio para inicializar los valores de las nuevas posiciones en el árbol MC.

- ***MC-RAVE***
  Utiliza el algoritmo *RAVE* (*Rapid Action Value Estimation*), que mejora el método original compartiendo los valores de diferentes subárboles del árbol MC, por ejemplo en el caso de las transposiciones.

[GS11] contiene información detallada sobre estas extensiones de MCTS.

Explicada la estrategia MCTS, se presentan a continuación los agentes que la usan; al igual que ocurrió con el método básico de Monte-Carlo se trata de dos agentes: uno con un número de simulaciones fijado de antemano y otro con un límite de tiempo.

Monte-Carlo Tree Search con número de simulaciones fijas

El primer agente que usa MCTS realiza un número de simulaciones determinado.

Partiendo del estado actual, ejecuta el ciclo de cuatro fases presentado anteriormente tantas veces como indique el número de simulaciones asignado. Establecer el número de simulaciones en MCTS es aún más difícil que para el agente Monte-Carlo básico porque ahora cada simulación necesita más tiempo para llevarse a cabo debido a que debe gestionar el árbol MC.

Por otro lado, el agente dispone de una opción que permite reutilizar el árbol MC de un movimiento para el siguiente, ya que los valores del árbol son válidos de un movimiento a otro. Esto permite mejorar la evaluación de los movimientos a medida que el agente juega la partida, a costa de incrementar los recursos necesarios (en memoria y tiempo) para manejar el árbol MC. Recordemos que el árbol se expande un nodo en cada simulación.

Este agente tiene también un parámetro perteneciente a la constante de exploración c que se vio en la fase de selección. El parámetro permite ajustar la estrategia de selección favoreciendo la exploración (valores próximos a 1) o la explotación (valores próximos a 0).

El último agente también usa MCTS para evaluar el mejor movimiento, pero esta vez dispone de un tiempo limitado para devolver el movimiento.

Monte-Carlo Tree Search con límite de tiempo

Este agente usa MCTS con un límite de tiempo para realizar las simulaciones, esto es, las iteraciones completas de cuatro fases de MCTS (selección, expansión, simulación y propagación).

Las iteraciones deben completarse totalmente para que sean válidas; de ahí que asignar el límite de tiempo no sea una tarea trivial. Si cada iteración necesita más tiempo del disponible, el tiempo límite del agente se extenderá hasta completar la última iteración. El límite de tiempo viene expresado en segundos al igual que el resto de jugadores con límite de tiempo que se han desarrollado.

Este agente también cuenta con la opción de reutilizar el árbol MC de un movimiento a otro; y también tiene parámetro para ajustar el nivel de exploración en la fase de selección.

Capítulo 4

Heurísticos

Este capítulo presenta los conceptos de heurístico y evaluador heurístico. Define la función de evaluación heurística que deben implementar los evaluadores heurísticos que necesitan las estrategias presentadas en el capítulo 3. Se describen dos evaluadores heurísticos que implementan la función de evaluación de forma genérica, mediante tablas de valores y redes neuronales; y se estudiará una forma de entrenarlos con aprendizaje con refuerzo mediante el método de las diferencias temporales. Estos evaluadores serán en la medida de lo posible, independientes del juego pues no necesitarán información sobre el mismo. Por último, se presentarán también los evaluadores heurísticos desarrollados específicamente para los juegos del Conecta-4 y del Go.

Un **heurístico** es una función o algoritmo que nos ayuda a encontrar soluciones "buenas" a los problemas. El heurístico proporciona información que nos permite decidir entre varias alternativas, aunque esta información puede estar incompleta o no ser completamente fiable. Los heurísticos generalmente se emplean en problemas donde no se puede obtener una solución óptima bajo los requisitos dados de tiempo y espacio. En esos casos nos conformamos con una buena solución.

Un **evaluador heurístico** nos permite evaluar si una situación dada nos es favorable o no, usando para ello una función de evaluación heurística. El agente presentado en 3.5.2 al que llamábamos, valga la redundancia, *Agente evaluador heurístico*, usa un evaluador heurístico para decidir el próximo movimiento.

A continuación se define la función de evaluación heurística, cómo implementar esta función dependerá de cada juego; aunque también se puede implementar de manera genérica para cualquier juego como se verá en las siguientes secciones.

4.1. Función de evaluación heurística

Este apartado describe las características que deben cumplir las funciones de evaluación heurísticas, de forma que permitan a la estrategia que las use cumplir su objetivo.

Una función de evaluación devuelve una estimación de la utilidad esperada de una posición dada del juego, independientemente de si se trata de una posición final o no. Por un lado, la función de evaluación debería ordenar los estados terminales del mismo modo que la función de utilidad verdadera del juego. Por otro lado, para estados no terminales la función de evaluación debería estar fuertemente correlacionada con las posibilidades reales de ganar. En último lugar, el cálculo no debe emplear demasiado tiempo, ya que la función será invocada repetidas veces por la estrategia para decidir un único movimiento.

La función de evaluación heurística empleada para evaluar los estados de un juego debe devolver un valor positivo si el estado es favorable para nuestro jugador (que llamaremos *MAX* siguiendo la terminología de minimax introducida en 3.5.3), un valor negativo si el estado es desfavorable y cero si el estado es indiferente (no es favorable ni desfavorable). Este valor será más grande cuanto más favorable sea el estado para *MAX* y más pequeño cuanto más desfavorable sea. Llamaremos a la función de evaluación *e(n)*, donde *n* es el estado a evaluar:

$$e(n) = \begin{cases} \infty & \text{si } n \text{ es un estado terminal y gana } MAX \\ > 0 & \text{si } n \text{ es favorable para } MAX \\ 0 & \text{si } n \text{ es indiferente (“empate”)} \\ < 0 & \text{si } n \text{ es desfavorable para } MAX \\ -\infty & \text{si } n \text{ es un estado terminal y pierde } MAX \end{cases} \tag{4.1}$$

Dos formas de implementar esta función de manera genérica (sin necesitar conocimiento del juego en cuestión) son las tablas de valores y las redes neuronales que darán lugar a nuestros dos evaluadores genéricos. Para ello es necesario que dispongan de algún tipo de aprendizaje que les permita aprender cuándo un estado es favorable o no.

A continuación se presenta el aprendizaje con refuerzo mediante el método de las diferencias temporales, que será la forma de entrenar a los evaluadores con tablas de valor y redes neuronales.

4.2. Aprendizaje con refuerzo

El **aprendizaje con refuerzo** es un método de aprendizaje automático en el que el algoritmo aprende observando el medio que le rodea. La información de entrada es el *feedback* o retroalimentación que obtiene del medio como respuesta a sus acciones. Por lo tanto, el sistema aprende por sí mismo a base de ensayo-error. Un agente que use aprendizaje con refuerzo necesita saber que algo bueno ha ocurrido cuando gana y que algo malo ha ocurrido cuando pierde. Esta clase de retroalimentación se denomina **recompensa** (*reward*).

En este tipo de aprendizaje el agente no tiene ningún conocimiento a priori sobre el entorno, ya que no es necesario. Lo único que tiene que saber el agente es cuándo ha ganado o ha perdido y puede usar esta información para aprender una función de evaluación que proporcione

estimaciones certeras y razonables de la probabilidad de ganar a partir de una situación dada.

En nuestro caso no entrenaremos directamente al agente sino al evaluador heurístico que usa. El entrenamiento se realizará mediante el método de las diferencias temporales, que se explica a continuación.

4.2.1. Método de las diferencias temporales

Llamaremos función de recompensa a la función que indica al evaluador heurístico cuándo ha ganado, ha empatado o ha perdido; y función de utilidad a la función de evaluación que debe aprender y que proporciona la recompensa esperada a partir de una situación dada.

La función de recompensa (R) define el objetivo del problema: obtener la mayor recompensa posible. Dado un estado, devuelve su valor de recompensa real. Este valor puede ser por ejemplo un número positivo en caso de que el estado sea ganador, negativo y opuesto si es perdedor y cero si es empate.

La función de utilidad (U) define lo "deseable" que es un estado, es decir, su propio valor de recompensa más la utilidad esperada de sus estados sucesores. Dado un estado, devuelve la recompensa esperada a largo plazo. La acción o movimiento a elegir en cada momento es la que lleva a un estado de mayor utilidad.

El **método de las diferencias temporales** (*TD*) consiste en usar las acciones realizadas para ajustar los valores de los estados correspondientes según los valores de recompensa obtenidos. Se elaboran unas secuencias de entrenamiento, en este caso las secuencias de entrenamiento serán partidas completas de un juego; y partiendo de unos valores iniciales cualesquiera, en cada paso (estado $n \rightarrow$ sucesor s) de cada secuencia se actualiza el valor de utilidad correspondiente:

$$U(n) = U(n) + \alpha(R(n) + U(s) - U(n))$$

donde α es la tasa de aprendizaje, expresada como un escalar en el intervalo $(0,1)$. La diferencia de utilidades entre estados sucesivos es la regla de aprendizaje por diferencias temporales y los valores así calculados convergen a las utilidades óptimas de los estados.

En el caso de los juegos el valor inicial de recompensa para un estado es cero. Por lo que si el valor de utilidad del estado sucesor es menor que el valor del padre, el valor de este último desciende; por el contrario, si el valor de utilidad del estado sucesor es mayor que el valor del padre, el valor de este último aumentará.

Para garantizar que se obtiene una buena estrategia, se pueden realizar movimientos exploratorios durante el entrenamiento. Para ello se introduce una componente aleatoria para los movimientos del evaluador a entrenar: cada movimiento tendrá una probabilidad de ser aleatorio (exploratorio); en caso de que el movimiento sea exploratorio, no se entrenará al evaluador.

El método de las diferencias temporales se puede aplicar a cualquier juego, pues no necesita de un modelo concreto para llevar a cabo sus actualizaciones; el entorno proporciona las conexiones entre los diferentes estados en forma de movimientos posibles. Este tipo de apren-

dizaje es sencillo y no requiere de mucho tiempo de cálculo, pero tiene la desventaja de que el aprendizaje el lento.

Las siguientes secciones describen los evaluadores que implementan la función de evaluación de forma genérica, y que son entrenados empleando aprendizaje con refuerzo mediante el método de las diferencias temporales.

4.3. Tabla de Valor

El primer evaluador heurístico que presentamos es la tabla de valor. Este evaluador implementa la función de evaluación mediante una tabla que almacena el valor de utilidad para cada estado. El evaluador es entrenado mediante el método de las diferencias temporales; actualizando los valores de la tabla en cada paso del entrenamiento:

$$tabla(n) \leftarrow tabla(n) + \alpha(tabla(s) - tabla(n))$$

donde n es el estado cuyo valor se actualiza, s es el sucesor del estado n elegido y α es la tasa de aprendizaje.

Los valores de utilidad de cada estado al final del entrenamiento se corresponden con el valor de evaluación que devolverá el evaluador para cada estado:

$$e(n) = tabla(n)$$

La ventaja de este evaluador es que es independiente del juego y podrá usarse en cualquiera de los juegos propuestos. La principal desventaja es la propia tabla de valores, debido al consumo de memoria.

4.4. Red Neuronal

El segundo evaluador heurístico es una red neuronal. Este evaluador implementa la función de evaluación mediante una red neuronal multicapa con el algoritmo de aprendizaje mediante retropropagación.

Para diseñar la arquitectura de la red, se seleccionan una serie de características del entorno. El número de neuronas de entrada dependerá de las características elegidas, es decir, de cómo se codifiquen los estados de los juegos; lo que implica que este evaluador no pueda ser del todo independiente del juego.[1]

La red tiene dos neuronas de salida para realizar la evaluación. La primera (p_1) se interpreta como la probabilidad de que la posición sea ganadora para el primer jugador; y la segunda (p_2)

[1] En 4.5.2 y 4.6.4 se definen la funciones de codificación propuestas para los estados del juego del Conecta-4 y del Go.

como la probabilidad de que la posición sea ganadora para el segundo jugador. El resultado de la evaluación es la diferencia entre las dos salidas:

$$e(n) = p_1 - p_2$$

donde n es el estado evaluado.

La red cuenta con una sola capa intermedia de neuronas. El número de neuronas de esta capa será un parámetro adicional de la red. La función que cumple dicha capa intermedia es tratar de realizar una proyección en la que resulten separables linealmente los patrones de entrada de manera que las unidades de salida pueda realizar una clasificación correcta. Establecer de antemano el número de neuronas intermedias no es sencillo y puede ser necesario realizar varios experimentos antes de encontrar la estructura ideal de la red para un determinado juego.

La forma de entrenar una red neuronal multicapa mediante aprendizaje por refuerzo es un algoritmo conocido como $TD(\lambda)$ (nosotros usaremos la versión simplificada del método TD, visto en el apartado 4.2.1); entrenando la red del siguiente modo: cuando una posición del juego es final, se entrena con la salida correcta (posición ganadora, perdedora o de empate para el jugador); cuando la posición no sea final, se entrena la red para que su evaluación sea parecida a la de la siguiente posición en la secuencia de entrenamiento.

La red dispone también de dos parámetros adicionales para tener un mayor control sobre su aprendizaje. Por un lado tiene una *tasa de aprendizaje* que permite ajustar la velocidad a la que aprende la red. Por otro lado tiene un *momento* de entrenamiento que indica la influencia que tendrá la iteración anterior sobre la actual. Ambos valores son porcentajes expresados mediante escalares en el intervalo $[0,1]$.

4.5. Heurísticos para el Conecta-4

En esta sección se define una función heurística que proporciona una evaluación de los estados del juego para el Conecta-4, empleando para ello una matriz de posibilidades. También se presenta una función de codificación de los estados del juego para un evaluador con red neuronal.

4.5.1. Matriz de posibilidades

Dado un estado del juego la función de evaluación heurística debe devolver un número positivo si el estado es favorable para nuestro jugador, un número negativo si el estado es desfavorable y cero si es indiferente.

Llamaremos a nuestro jugador *MAX* y a su oponente *MIN*. La función de evaluación para los estados del juego Conecta-4 se define como: *número de posibilidades de conectar 4 fichas que tiene MAX, menos el número de posibilidades de conectar 4 fichas que tiene MIN.*

Para determinar el número de posibilidades de conectar 4 fichas se emplea una *matriz de posibilidades*. A continuación se detalla un método para construir la matriz de posibilidades. Por simplicidad y sin perder genericidad se considera una versión simplificada del juego: el Conecta-3 sobre un tablero 4x4.

Cada posición del tablero tiene un número de posibilidades distintas de conectar 3 fichas. La figura 4.1 muestra las posibilidades de conectar 3 fichas que tiene la posición correspondiente a la esquina inferior izquierda del tablero. Esta posición interviene en tres posibilidades que numeramos del 0 al 2. La figura 4.2 muestra las posibilidades de conectar 3 fichas que tiene la posición siguiente. En este caso también tiene tres posibilidades que numeramos del 3 al 5.

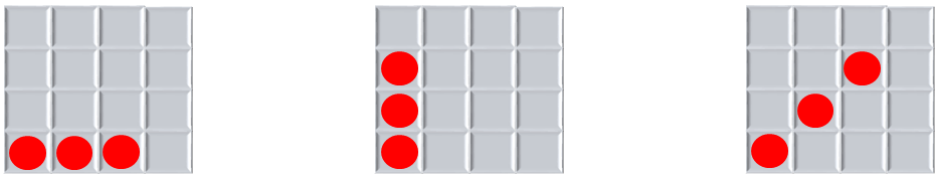


Figura 4.1: Posibilidades 0, 1 y 2.

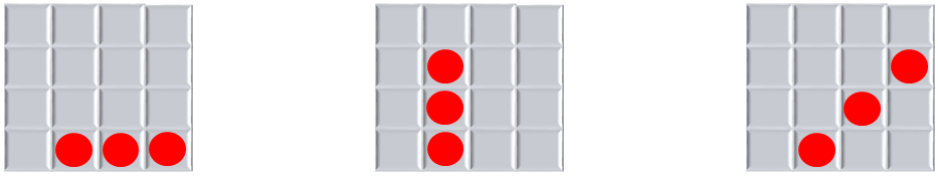


Figura 4.2: Posibilidades 3, 4 y 5.

Recorriendo todas las posiciones del tablero y enumerando todas las posibilidades en las que interviene cada posición del tablero se obtiene una matriz de posibilidades como la mostrada en la figura 4.3.

9, 20, 21	12, 20, 22, 23	10, 14, 20, 22	13, 15, 22
1, 9, 16, 17	4, 10, 12, 16, 18, 19, 21	2, 6, 13, 14, 16, 18, 23	5, 7, 15, 18
1, 8, 9, 10	2, 4, 8, 11, 12, 13, 17	5, 6, 8, 11, 14, 19, 21	7, 11, 15, 23
0, 1, 2	0, 3, 4, 5	0, 3, 6, 17	3, 7, 19

Figura 4.3: Matriz de posibilidades.

Existen 24 posibilidades de conectar 3 fichas. Las posiciones menos interesantes del tablero corresponden a las cuatro esquinas. Las posiciones situadas en las columnas centrales, por el contrario, son las que ofrecen más posibilidades.

Para realizar la evaluación, la función heurística tiene acceso a la matriz de posibilidades del tablero. Cada jugador dispone de un vector de longitud n, donde n es el número de posibilidades

de conectar k fichas en el tablero. Para cada casilla ocupada en el tablero se actualizan los vectores adecuadamente con la ayuda de la matriz de posibilidades: si una casilla está ocupada por una ficha de *MAX*, se elimina del vector de *MIN* todas las posibilidades en las que interviene; y si está ocupada por una ficha de *MIN*, se elimina las posibilidades del vector de *MAX*. El valor devuelto por la función heurística es la diferencia entre la suma de posibilidades libres en el vector de *MAX* y la suma de posibilidades libres en el vector de *MIN*:

$$e(n) = posibilidades_{MAX}(n) - posibilidades_{MIN}(n)$$

A continuación se presenta el evaluador con red neuronal para el Conecta-4, definiendo la función para codificar los estados como entrada a la red.

4.5.2. Red neuronal para el Conecta-4

El evaluador con red neuronal presentado en la sección 4.4 necesitaba de una función de codificación de los estados de los juegos para usarlos como entrada de la red neuronal.

Para los estados del juego del Conecta-4 se codificará cada posición del tablero mediante dos entradas a la red neuronal:

- La primera entrada será 1 si hay una ficha del primer jugador y 0 en caso contrario.
- La segunda entrada será 1 si hay una ficha del segundo jugador y 0 en caso contrario.

De este modo, para un tablero *nxm* se necesitan *2xnxm* entradas en la red neuronal.

El número de neuronas de la capa intermedia no está fijado; lo que permite realizar varias pruebas con diferentes valores hasta encontrar el valor que mejor resultados obtenga.

Una vez definidos los heurísticos específicos para el juego del Conecta-4, presentamos los heurísticos para el juego del Go.

4.6. Heurísticos para el Go

En esta sección se estudiarán varios heurísticos para el juego del Go que permitan decidir si un estado del juego es mejor que otro.

Las funciones heurísticas están basadas en el objetivo principal del juego: controlar el mayor número de territorios del tablero, entendiendo por territorios el número de intersecciones libres conquistadas por un jugador. Ubicar piedras juntas ayuda a protegerlas entre sí y evitar ser capturadas. Por otro lado, colocar piedras de forma separada hace que se tenga influencia sobre una mayor porción del tablero. Parte de la dificultad estratégica del juego surge a la hora de encontrar un equilibrio entre estas dos alternativas. Los jugadores pueden jugar tanto de manera

ofensiva como defensiva y deben elegir entre tácticas de urgencia y planes a largo plazo más estratégicos.

Teniendo esto en cuenta se definen tres evaluadores heurísticos: uno basado únicamente en los territorios conquistados por cada jugador, y los otros dos en función de los puntos conseguidos por cada jugador hasta el momento, según las reglas de puntuación japonesas para el primer evaluador y según las reglas chinas para el segundo evaluador.

4.6.1. Evaluador de territorios

El primer evaluador tiene en cuenta solamente los territorios conquistados por cada jugador. El valor devuelto por la función heurística para un estado n es la diferencia entre el número de territorios pertenecientes a *MAX* y el número de territorios de *MIN*:

$$e(n) = territorios_{MAX}(n) - territorios_{MIN}(n)$$

4.6.2. Evaluador de puntos JP

El segundo evaluador se basa en la puntuación de cada jugador en el estado actual, según las reglas de puntuación japonesas. Recordemos que los japoneses cuentan un punto por cada intersección vacía dentro del territorio conquistado menos un punto por cada piedra que haya capturado el enemigo. El valor de la función heurística para un estado n es la diferencia entre los puntos de *MAX* y de *MIN* según las reglas de puntuación japonesas:

$$e(n) = puntos^{JP}_{MAX}(n) - puntos^{JP}_{MIN}(n)$$

4.6.3. Evaluador de puntos CH

El último evaluador propuesto es igual que el anterior pero teniendo en cuenta las reglas de puntuación chinas. Los jugadores chinos cuentan un punto por cada intersección vacía dentro del territorio más un punto por cada ficha del jugador sobre el tablero. Para un estado n, el valor de la función heurística es la diferencia de puntos entre *MAX* y *MIN* según las reglas de puntuación chinas:

$$e(n) = puntos^{CH}_{MAX}(n) - puntos^{CH}_{MIN}(n)$$

La siguiente sección presenta el evaluador con red neuronal para el Go, definiendo la función para codificar los estados como entrada a la red.

4.6.4. Red neuronal para el Go

Se define a continuación la función de codificación para los estados del Go que necesita la red neuronal como entrada. El evaluador con red neuronal se presentó en la sección 4.4 de

manera general.

La forma de codificar los estados será la misma que la empleada en el juego del Conecta-4. Cada posición del tablero se codifica mediante dos entradas a la red neuronal:

- La primera entrada será 1 si hay una ficha del primer jugador y 0 en caso contrario.
- La segunda entrada será 1 si hay una ficha del segundo jugador y 0 en caso contrario.

Para un tablero de *nxn* se necesitan *2xnxn* entradas en la red neuronal.

El número de neuronas de la capa intermedia tampoco está fijado para la red neuronal del Go porque puede ser útil modificar este número en función del tamaño del tablero de juego para encontrar la red neuronal ideal que juegue al Go.

Capítulo 5

Especificación

En este capítulo se detallan los requisitos del proyecto y se describen los casos de uso de la aplicación interactiva.

El objetivo del proyecto es construir un entorno interactivo que permita jugar y comparar las diferentes estrategias de juego en IA.

5.1. Requisitos

El proyecto puede dividirse en dos partes bien diferenciadas: por un lado el módulo de razonamiento de los agentes junto con los juegos; por otro lado la aplicación interactiva, es decir, la interfaz gráfica de usuario. Teniendo esto en cuenta se pueden extraer los requisitos de ambas partes de forma separada.

A continuación se presentan primero las características del módulo de razonamiento y después la aplicación gráfica de usuario.

5.1.1. Módulo de razonamiento

El proyecto incluye un módulo de razonamiento para los agentes jugadores y los juegos.

Los algoritmos de decisión deben ser independientes de los propios juegos, es decir, en cada juego debe poder usarse todos los algoritmos disponibles. Esto es complicado pues hay estrategias que necesitan de información concreta sobre el estado de los juegos para evaluar las posiciones. Para solucionarlo se separarán también los evaluadores heurísticos de los algoritmos que los usan, independizando totalmente los algoritmos de los juegos.

Los juegos tienen las siguientes características: juegos de dos jugadores, por turnos, de suma cero, de información perfecta y deterministas. El módulo se completará con la implementación de dos juegos:

- El juego del Conecta-4; considerando una versión generalizada del mismo con un tamaño de tablero *nxm* y una longitud ganadora de *k* fichas. Las pruebas se realizarán sobre

tableros de dimensiones 4x4 y 6x7 con longitudes ganadores de tres y cuatro fichas respectivamente.

- El juego del Go; considerando una versión generalizada del mismo con un tamaño de tablero *nxn*. Las pruebas se realizarán sobre un tablero de dimensiones 9x9.

Ambos juegos deben ser representados mediante un espacio de estados que entiendan todos los algoritmos.

Por otro lado, los algoritmos o estrategias deben ajustarse a un espacio de estados genérico, independiente del juego. Se tratará de versiones generales de los algoritmos, de forma que resulten sencillas de entender. Los jugadores que se desarrollarán son los presentados a continuación cuyas respectivas estrategias han sido estudiadas en el capítulo 3:

- Un jugador humano, que será el único jugador totalmente dependiente del juego, pues es necesario que pida el movimiento a realizar al usuario mediante un dispositivo de entrada como el teclado o el ratón.
- Un jugador con estrategia aleatoria.
- Un jugador con evaluador heurístico.
- Un jugador con estrategia minimax y una profundidad máxima de búsqueda fijada.
- Un jugador con estrategia minimax y un tiempo limitado para realizar la búsqueda.
- Un jugador con poda alfa-beta y una profundidad máxima de búsqueda.
- Un jugador con poda alfa-beta y un tiempo limitado.
- Un jugador con estrategia minimax y una profundidad máxima de búsqueda que incluye además una tabla de transposición.
- Un jugador con estrategia minimax y un tiempo limitado que incluye además una tabla de transposición.
- Un jugador con poda alfa-beta y una profundidad máxima de búsqueda que incluye también una tabla de transposición.
- Un jugador con poda alfa-beta y un tiempo limitado que incluye también una tabla de transposición.
- Un jugador que utiliza el método de Monte-Carlo para decidir el mejor movimiento realizando un número determinado de simulaciones.
- Un jugador que utiliza el método de Monte-Carlo con un límite de tiempo para realizar las simulaciones.

- Un jugador que utiliza el método Monte-Carlo Tree Search para decidir el mejor movimiento realizando un número determinado de simulaciones.
- Un jugador que utiliza el método Monte-Carlo Tree Search con un límite de tiempo para realizar las simulaciones.

La tabla 5.1 muestra de forma resumida todas las estrategias junto con los parámetros permitidos.

Tabla 5.1: Estrategias y sus parámetros.

Estrategias	**Parámetros**			
	Prof. máx. búsqueda	Nº simulaciones	Límite tiempo	Heurísticos
Aleatoria				
Evaluador heurístico				✓
Minimax	✓		✓	✓
Alfa-Beta	✓		✓	✓
Minimax (tabla de transposición)	✓		✓	✓
Alfa-Beta (tabla de transposición)	✓		✓	✓
Monte-Carlo		✓	✓	
Monte-Carlo Tree Search		✓	✓	

Los evaluadores heurísticos que necesitan algunos de los jugadores son independientes de estos últimos como se comentó anteriormente. Los evaluadores heurísticos que se incluirán son:

- Un evaluador para el Conecta-4 basado en una matriz de posibilidades.
- Un evaluador para el Go basado el número de territorios conquistados.
- Un evaluador para el Go basado en los puntos según las reglas japonesas.
- Un evaluador para el Go basado en los puntos según las reglas chinas.
- Un evaluador que emplea una tabla de valor.
- Un evaluador que emplea una red neuronal.

Los dos últimos evaluadores (tabla de valor y red neuronal) necesitan de un entrenamiento previo; por lo que se incluirá un módulo de aprendizaje que permita entrenar dichos evaluadores.

El módulo de razonamiento debe proporcionar un marco de trabajo de forma que permita modificar o añadir nuevos algoritmos, juegos y heurísticos de manera independiente y sencilla.

A continuación se detallan los requisitos de la aplicación interactiva que hará uso del módulo de razonamiento de los algoritmos y juegos.

5.1.2. Aplicación interactiva

El proyecto incluye una aplicación interactiva basada en una interfaz gráfica de usuario. La aplicación integra el módulo de razonamiento de los algoritmos y juegos dando acceso a toda su funcionalidad.

La aplicación permite al usuario seleccionar un juego disponible y configurar sus parámetros, como el tamaño del tablero. También permite seleccionar la estrategia a usar por cada jugador, configurando a su vez los parámetros de la estrategia en cuestión, como por ejemplo los evaluadores heurísticos o el tiempo disponible para realizar la jugada. En el caso de los evaluadores heurísticos que lo necesiten, la aplicación ofrece la posibilidad de entrenarlos configurando los diferentes aspectos del entrenamiento (número de partidas, oponente, pruebas, etc.).

Una vez seleccionado el juego y los jugadores, la aplicación permite jugar, ver el desarrollo de una partida, simular varias partidas o analizar estados. Al finalizar cada una de estas acciones, la aplicación muestra estadísticas sobre las mismas y ofrece la opción de generar un informe con las estadísticas.

La aplicación también cuenta con un sistema de ayuda que proporciona información sobre los componentes visuales cuando el usuario se sitúa sobre ellos.

A partir de estos requisitos, la siguiente sección detalla los principales casos de uso de la aplicación.

5.2. Casos de uso

La aplicación interactiva cuenta principalmente con tres casos de uso: jugar, simular y analizar estado, que se muestran en el diagrama de la figura 5.1. El actor principal de todos los casos de uso es el usuario de la aplicación.

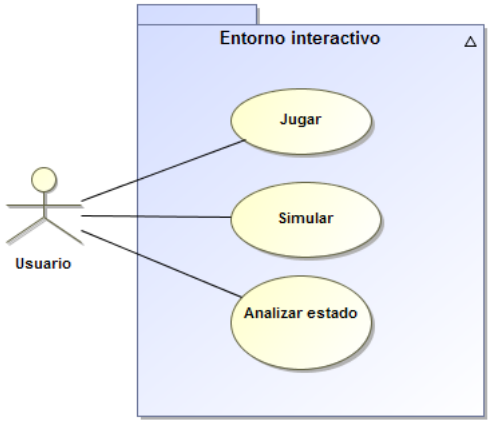


Figura 5.1: Diagrama de los principales casos de uso de la aplicación.

1. **Caso de uso:** Jugar

   El usuario juega una partida al juego elegido con los jugadores seleccionados.

   Extensión: Si ninguno de los jugadores es humano, se muestra el desarrollo de la partida entre los jugadores controlados por el ordenador.

2. **Caso de uso:** Simular

   El usuario indica un número de partidas y la aplicación juega las partidas entre los dos jugadores seleccionados. El desarrollo de las partidas no se muestra.

   Precondición: Ambos jugadores deben ser controlados por el ordenador, en caso contrario la opción de simular estará desactivada.

3. **Caso de uso:** Analizar estado

   El usuario crea un estado concreto del juego elegido, colocando manualmente las fichas sobre el tablero. A partir de ese estado, los jugadores seleccionados realizan independientemente un único movimiento cada uno.

   Precondición: Ambos jugadores deben ser controlados por el ordenador, en caso contrario la opción de analizar estado estará desactivada.

Cada uno de estos casos de uso produce unas estadísticas diferentes que son mostradas al finalizar los mismos. Parte de esta información depende del jugador seleccionado; cada estrategia proporciona sus propias estadísticas, que son independientes del caso de uso elegido. La aplicación permite además generar un informe en texto plano con toda la información de las estadísticas obtenidas. Una vez terminado un caso de uso, los jugadores se pueden reutilizar para iniciar otro caso de uso sin tener en cuenta las estadísticas anteriores; lo que permite realizar tantas pruebas como se desee.

A continuación se presentan los casos de uso pertenecientes a la selección y configuración de los juegos y de las estrategias. Para los juegos tenemos un único caso de uso (seleccionar juego) con la opción de modificar sus parámetros. La figura 5.2 muestra el diagrama para este caso de uso.

4. **Caso de uso:** Seleccionar juego

   El usuario elige el juego deseado de entre los disponibles.

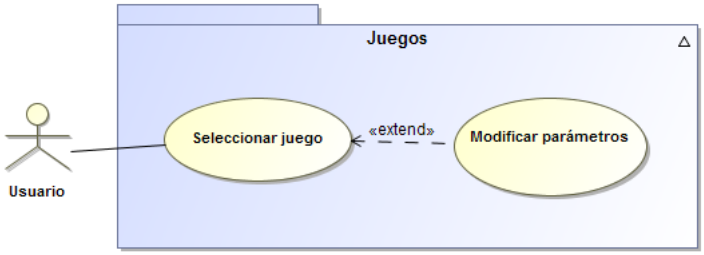


Figura 5.2: Diagrama de casos de uso para los juegos.

Extensión: El usuario configura el juego seleccionado. Los parámetros a configurar dependen del juego:

- Conecta-4: El usuario elige el tamaño del tablero y la longitud ganadora de fichas.
- Go: El usuario elige el tamaño del tablero, las reglas de puntuación y los puntos de ventaja para el segundo jugador.

En el caso de las estrategias, existen dos casos de uso: uno para seleccionar la estrategia del primer jugador y otro para seleccionar la estrategia del segundo jugador. El escenario principal de ambos casos de uso es idéntico, por lo que sólo se describe uno de forma general. El diagrama de la figura 5.3 muestra el caso de uso de forma genérica para ambos jugadores.

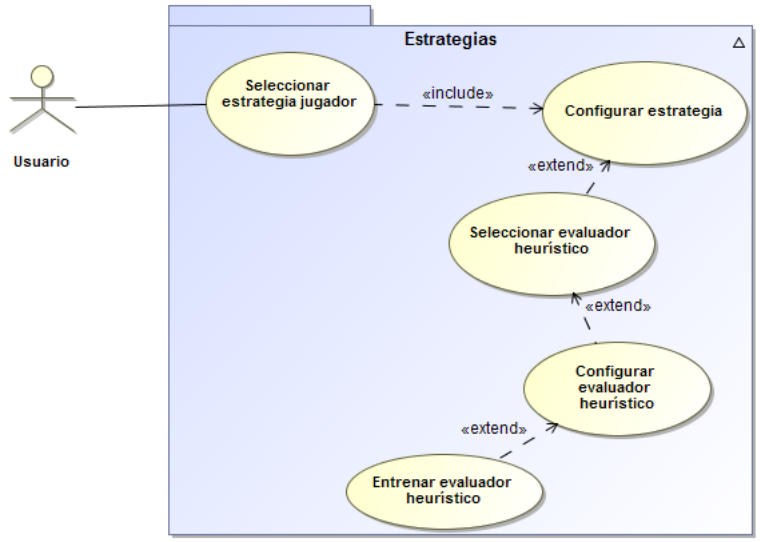


Figura 5.3: Diagrama de casos de uso para las estrategias.

5. **Caso de uso:** Seleccionar estrategia del jugador

   El usuario selecciona y configura los parámetros de la estrategia para uno de los jugadores.

   Precondición: El usuario ha seleccionado un juego.

   Extensiones:

   5a. El usuario selecciona un evaluador heurístico.

   5a.1. El usuario configura el evaluador heurístico seleccionado.

   5a.1a. El usuario configura el entrenamiento y la aplicación entrena el evaluador heurístico.

Capítulo 6

Arquitectura de la aplicación

Este capítulo presenta la arquitectura del sistema. Describe el diseño del módulo de razonamiento y muestra los diagramas de clases correspondientes. También se presentan brevemente el diseño de la interfaz de la aplicación con los patrones de diseño usados en la misma.

6.1. Módulo de razonamiento

El módulo de razonamiento se ha dividido en tres módulos más pequeños: el módulo de los juegos, el de los jugadores o estrategias y el de los heurísticos. Todos los módulos se encuentran relacionados mediante unas interfaces definidas. A continuación se describen en detalle cada uno de estos módulos.

6.1.1. Juegos

En el capítulo 2 se definió formalmente un juego como una clase de problemas de búsqueda que contiene: un estado inicial, una función sucesor, un test terminal y una función de utilidad o función objetivo. Definiendo una interfaz adecuada se puede construir un módulo que permita implementar fácilmente cualquier juego del tipo considerado.

La figura 6.1 muestra el diagrama de clases del módulo de juegos junto con los dos juegos considerados (Conecta-K y Go). Para simplificar el diagrama no se muestra la implementación de ambos juegos, sino únicamente la relación existente entre ellos y el módulo de juegos.

El elemento principal del módulo de juegos es la interfaz *`EstadoJuego`*, que proporciona una representación de los estados de un juego. Se trata del elemento más importante de todo el módulo de razonamiento: cualquier juego debe implementar esta interfaz para representar sus estados. La interfaz contiene una función sucesor (método `hijos`), un test terminal (`agotado`), y métodos que permiten indexar el estado en una posible estructura, conocer el ganador (si lo hay) y el jugador al que le toca jugar, comparar dos estados y proporcionar una descripción del estado.

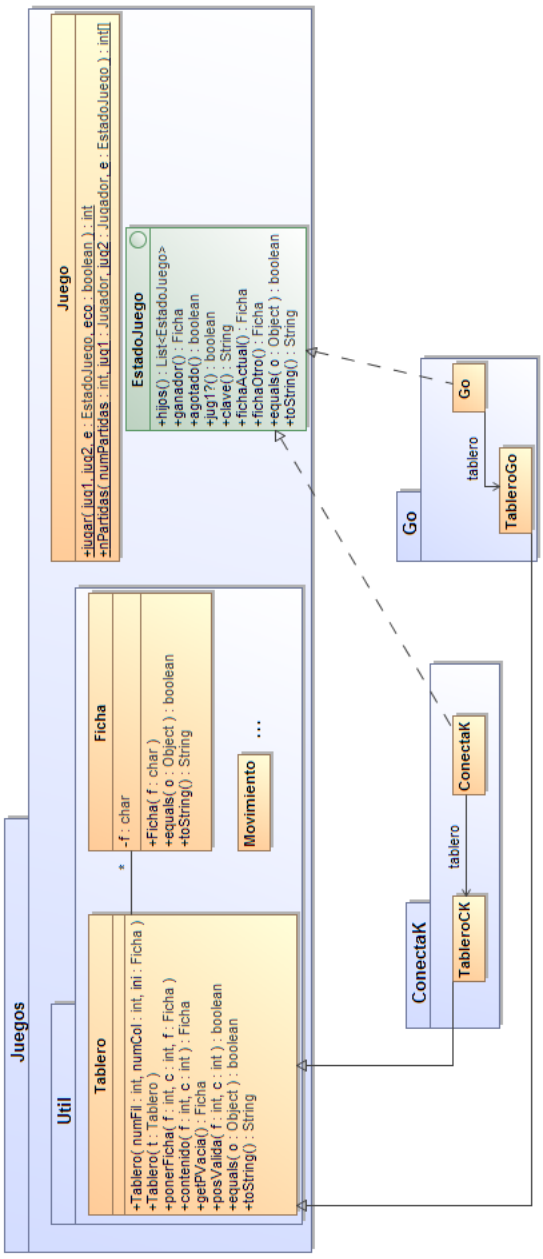


Figura 6.1: Diagrama de clases de los juegos.

Esta representación permite obtener el árbol de juegos a partir un estado gracias al método `hijos` que devuelve los sucesores inmediatos del estado actual.

Nótese que la interfaz no proporciona una función de utilidad para los estados terminales, ya que el método ganador sólo indica quién ha ganado. El valor de utilidad al final de una partida se ha diseñado de manera independiente del juego: el método estático `jugar` de la clase `Juego` permite jugar una partida entre dos jugadores a partir de un estado inicial y devuelve el valor de utilidad obtenido (1 si gana el primer jugador, -1 si gana el segundo jugador y 0 en caso de empate). Igualmente el método `nPartidas` juega un número determinado de partidas devolviendo los valores de utilidad de cada partida.

El módulo también proporciona un paquete de utilidades que ayuda a implementar los juegos como un tablero genérico, fichas o una representación de los movimientos.

El diagrama de la figura 6.1 sólo muestra en los paquetes `ConectaK` y `Go` las clases que representan a dichos juegos. Estos paquetes son independientes del módulo de juegos y contienen otras clases como evaluadores heurísticos específicos para cada juego o jugadores dependientes del juego como el jugador humano.

El siguiente apartado muestra el diseño del módulo correspondiente a las estrategias.

6.1.2. Estrategias

El módulo de estrategias se muestra en el diagrama de la figura 6.2.

Toda clase que represente un agente jugador debe implementar la interfaz *`Jugador`*. Esta interfaz solamente tiene un método (`mueve`) que devuelve el estado resultante de que el jugador mueva en el estado actual dado.

El `JugadorAleatorio` implementa esta interfaz de forma general lo que le permite jugar a cualquier tipo de juego. No ocurre lo mismo con el `JugadorHumano`, que necesita pedir el movimiento a realizar mediante un dispositivo de entrada, de ahí que se trate de una clase abstracta. Cada juego debe implementar su propio jugador humano.

El `JugadorEvaluar` necesita de un evaluador heurístico que proporciona la evaluación de los estados; toda estrategia que necesite de un heurístico se considera una especialización de este jugador. El resto de jugadores no forman parte del paquete `Estrategias` porque cada jugador puede implementar su estrategia de forma diferente, por ejemplo, la estrategia minimax se puede implementar siguiendo la definición 3.1 o mediante el algoritmo negamax (3.5.3). En el diagrama, por simplicidad, no aparecen todos las estrategias desarrolladas.

También se proporciona un paquete de utilidades para las estrategias que necesiten de una alarma de tiempo, una tabla hash o una función hash[1]. Para simplificar el diagrama tampoco se muestran explícitamente las asociaciones entre los jugadores y las utilidades usadas por cada

[1]La función hash implementada realiza una codificación mediante el algoritmo de reducción criptográfico *MD5*[Riv92].

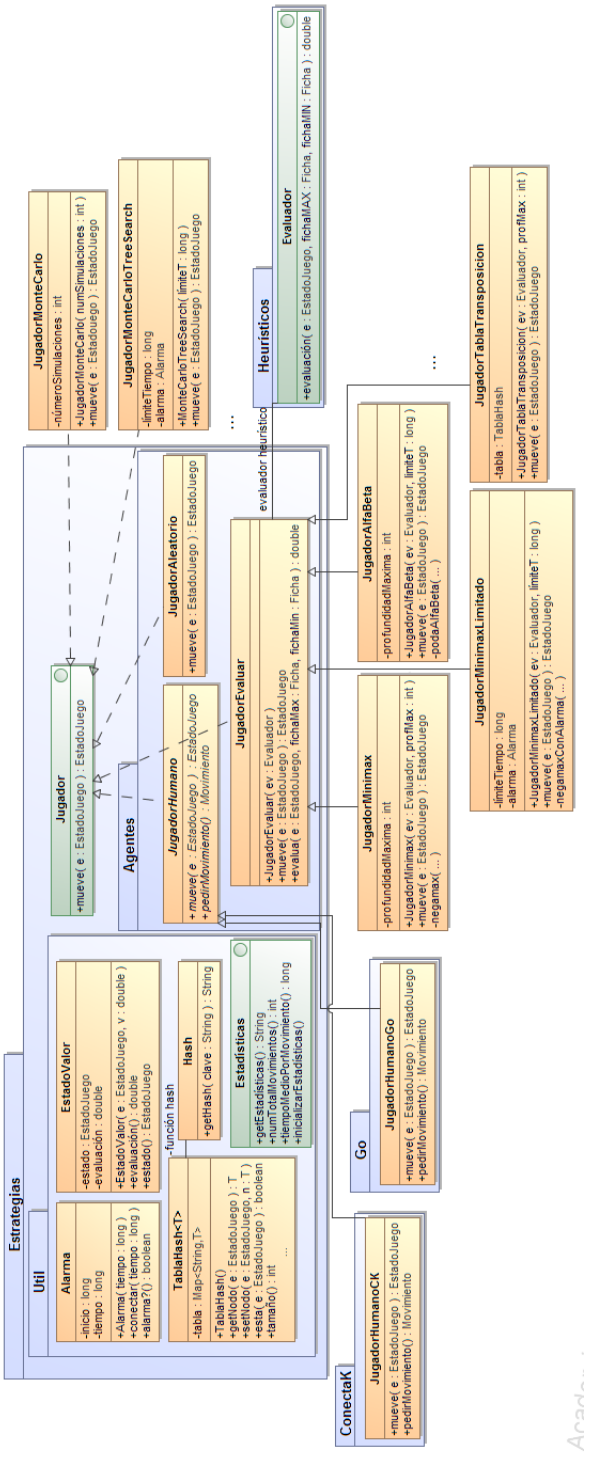


Figura 6.2: Diagrama de clases de las estrategias.

uno. Destaca la interfaz *Estadísticas* que deberán implementar las estrategias que deseen proporciona sus propias estadísticas.

A continuación se detalla el módulo de los heurísticos.

6.1.3. Heurísticos

El módulo de heurísticos contiene las definiciones de los objetos evaluadores y permite entrenarlos mediante aprendizaje por refuerzo. La figura 6.3 muestra en el diagrama de este módulo.

La interfaz *Evaluador* representa los objetos evaluadores heurísticos que proporcionan el valor de evaluación de un estado. El juego que quiera proporcionar un heurístico para sus estados debe implementar esta interfaz siguiendo la definición de la función de evaluación heurística dada en 4.1.

El módulo ofrece un paquete para definir evaluadores entrenables mediante aprendizaje por refuerzo; para ello los evaluadores deben implementar la interfaz *DiferenciasTemporales*. La clase `Entrenamiento` permite entrenar a los jugadores. Debido a que los valores aprendidos por el evaluador son diferentes según se trate del primer jugador o del segundo, la clase proporciona métodos diferentes para entrenar al primer jugador, al segundo jugador o a ambos jugadores simultáneamente; para simplificar el diagrama, no se muestran todos estos métodos, pero sus cabecera son idénticas a las mostradas teniendo en cuenta que el jugador que entrena debe ser un `JugadorEvaluar` o una especialización del mismo.

El módulo también proporciona dos evaluadores: las tablas de valor y la red neuronal, ambos entrenables. La red neuronal es una clase abstracta porque la función de codificación de los estados debe ser definida para cada juego. Para la red neuronal se ha usado una implementación de libre distribución llamada *Encog Machine Learning Framework* y que está disponible para varios lenguajes de programación [ENC08].

Los tres módulos juntos que se han presentado forman el marco de trabajo del proyecto.

6.1.4. Marco de trabajo

El marco de trabajo está compuesto por los tres módulos presentados en las secciones anteriores, concretamente los paquetes: `Juegos`, `Estrategias` y `Heurísticos`.

El marco de trabajo es una arquitectura especializada para el dominio de los juegos en IA; describe las interfaces, implementa algunos componentes y establece las reglas de interacción entre los componentes. No se trata de una biblioteca de clases puesto que el marco de trabajo contiene abstracciones que pueden instanciar o invocar a las abstracciones del desarrollador que extienda el marco.

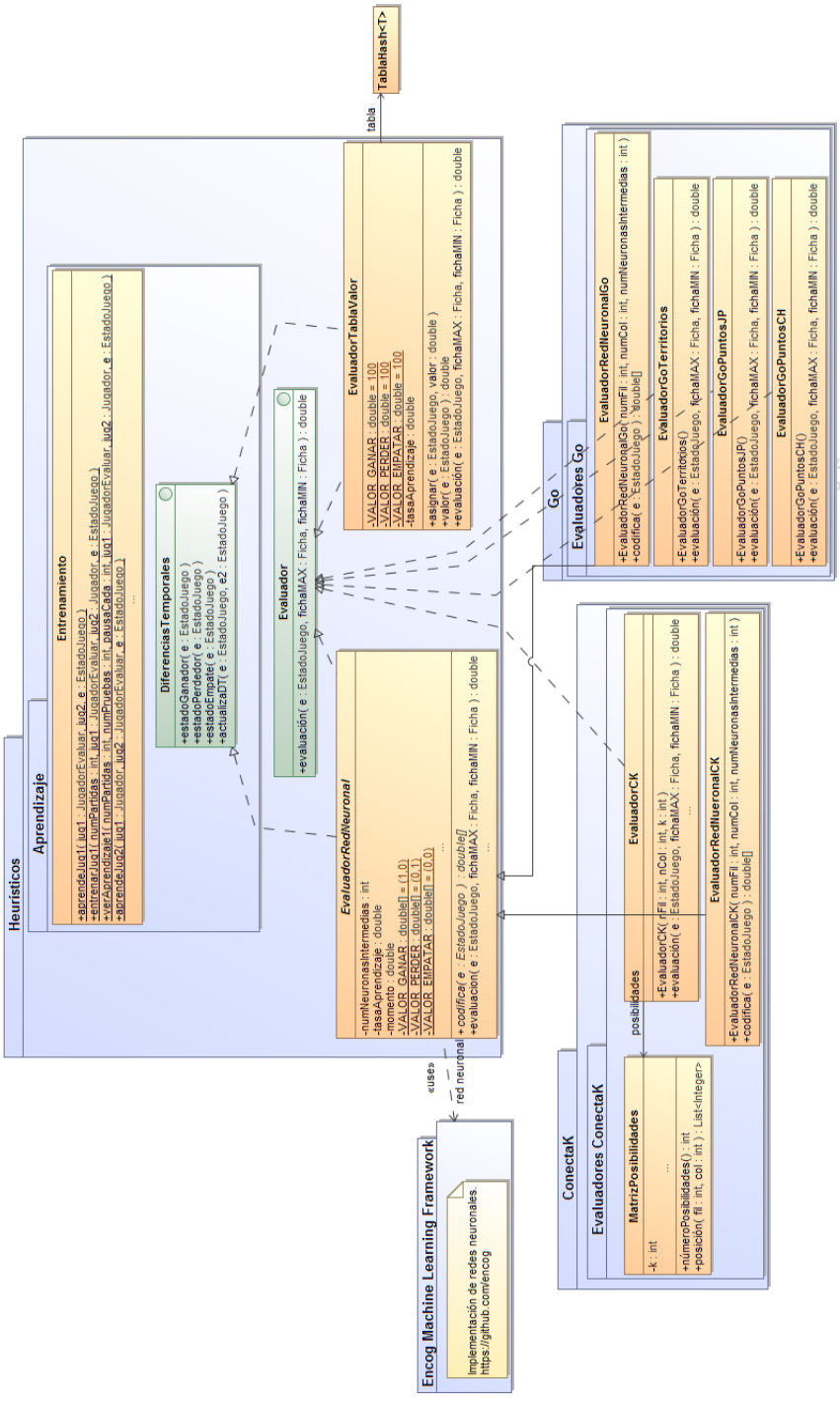


Figura 6.3: Diagrama de clases de los heurísticos.

Los diagramas de las figuras 6.1, 6.2 y 6.3 muestran también las extensiones del marco de trabajo, permitiendo extenderlo mediante herencia (extensión de caja blanca) o mediante composición (extensión de caja negra).

A continuación se describe la arquitectura de la aplicación interactiva que hace uso del módulo de razonamiento.

6.2. Aplicación interactiva

La aplicación interactiva es en realidad una interfaz de usuario para el módulo de razonamiento del sistema, que da acceso a toda su funcionalidad de forma fácil y atractiva.

Para diseñar la aplicación se ha seguido en todo momento el principio de integridad conceptual, por el cual los patrones generales de diseño del sistema se reflejan en cada parte del mismo. Un ejemplo es el sistema de ayuda de la aplicación, cuyo patrón se puede apreciar en cada parte de la interfaz. Lo mismo ocurre con el patrón *Modelo-Vista-Controlador* (*MVC*) aplicado independientemente a cada parte de la aplicación, aunque su uso no se refleje en la estructura global del sistema.

El diagrama de la figura 6.4 muestra las clases e interfaces que facilitan la incorporación de los juegos, estrategias y heurísticos a la interfaz gráfica.

Todos los juegos, estrategias y heurísticos desarrollados se han incorporado a la interfaz de la aplicación interactiva siguiendo el esquema mostrado. Cada elemento debe implementar su respectiva interfaz, lo que supone a su vez desarrollar los paneles de configuración adecuados, los controladores o el tablero para los juegos.

El núcleo de la interfaz gráfica consta de las vistas y los controladores necesarios para llevar a cabo los casos de uso de la aplicación, aunque estas clases no se muestran en el diagrama.

El sistema de información de ayuda está representado por la interfaz `InformaciónAyuda`, que es implementada por cada componente visual de la interfaz gráfica (pantallas y paneles) usando el patrón de diseño *Cadena de Mando* (*Chain of Responsability*). El usuario puede obtener información de ayuda sobre cualquier parte de la interfaz situándose sobre ella; la ayuda que se obtiene depende de la parte de la interfaz sobre la que solicita y de su contexto.

Por último, destacar que la aplicación se ha realizado íntegramente sobre tecnología *Java* [JAV] en su versión 1.6; aunque el diseño propuesto permite desarrollar el entorno interactivo al completo en cualquier lenguaje de programación orientado a objetos. La documentación en este proyecto es un aspecto clave, por lo que todo el código desarrollado se encuentra documentado en formato *javadoc* para que cualquier alumno, profesor o investigador pueda hacer uso de él.

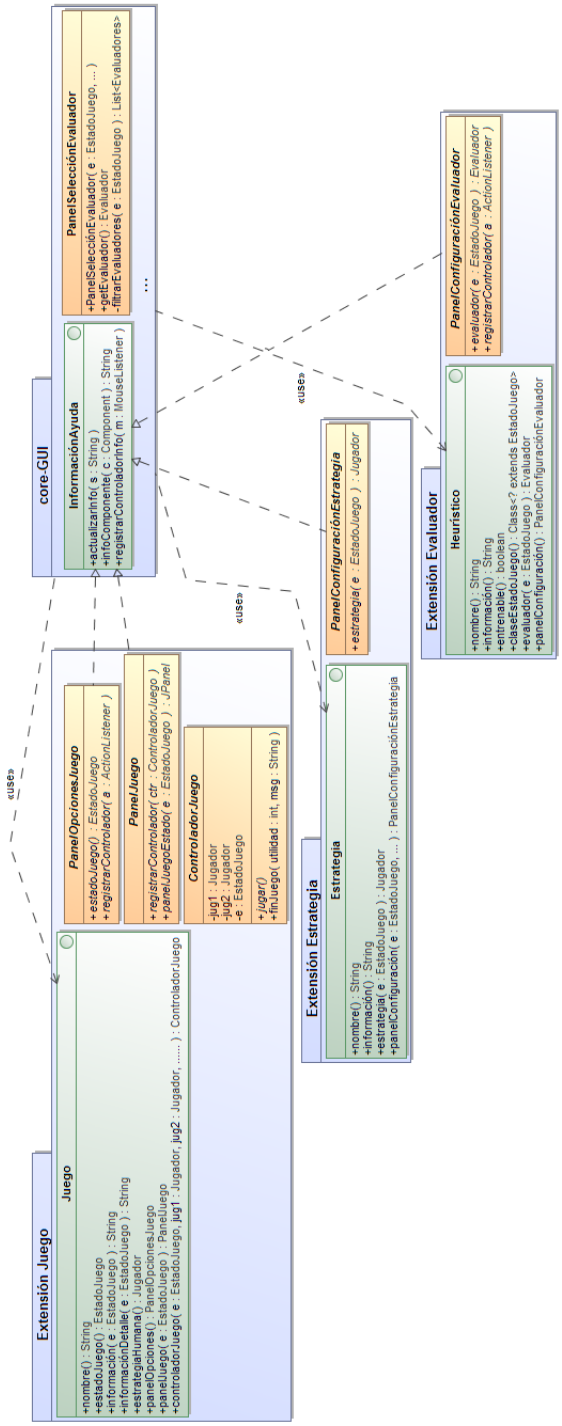


Figura 6.4: Diagrama de clases de las extensiones de la interfaz gráfica.

Capítulo 7

Experimentación

Este capítulo muestra algunas de las pruebas realizadas y los resultados obtenidos. Se comparan diferentes estrategias en los dos juegos desarrollados. Para ello se han realizado varias simulaciones y análisis de estados con las estrategias.

La siguiente sección se centra en las pruebas realizadas a los evaluadores heurísticos de cada juego, incluyendo algunas series de entrenamientos para los evaluadores entrenables.

7.1. Comparativa de evaluadores heurísticos

Primero se han comparado los evaluadores heurísticos de cada uno de los juegos de forma independiente.

7.1.1. Evaluadores del Conecta-4

Para comparar los evaluadores heurísticos del Conecta-4 se han jugado 100 partidas entre un jugador evaluador (`JugadorEvaluar`) y un jugador aleatorio para cada prueba. Recordemos que un `JugadorEvaluar` considera todos los movimientos inmediatos, los evalúa heurísticamente y escoge el mejor.

Las pruebas se han realizado en un tablero de dimensiones 6x7 con una longitud ganadora de cuatro fichas. Los evaluadores heurísticos usados son la matriz de posibilidades (número de posibilidades de conectar cuatro fichas de *MAX* menos el número de posibilidades de conectar cuatro fichas que tiene *MIN*), la tabla de valor y la red neuronal. La tabla de valor y la red neuronal se han entrenado con 2.000 partidas frente a un jugador aleatorio, con una tasa de aprendizaje de 0.1 y una tasa de exploración de 0.1. La red neuronal cuenta con una única neurona en la capa intermedia.

Las tablas 7.1 y 7.2 presentan los resultados de las pruebas realizadas con el `JugadorEvaluar` jugando como primer jugador y como segundo jugador respectivamente. Las figuras 7.1 y 7.2 muestran de forma gráfica estos resultados.

Tabla 7.1: Comparativa de los evaluadores heurísticos del Conecta-4 para el primer jugador.

Jugador 1:	**Ev. Heur. (Matriz de posibilidades)**	**Tabla de valor**	**Red neuronal**
Gana	98 %	84 %	88 %
Empata	0 %	0 %	0 %
Pierde	2 %	16 %	12 %

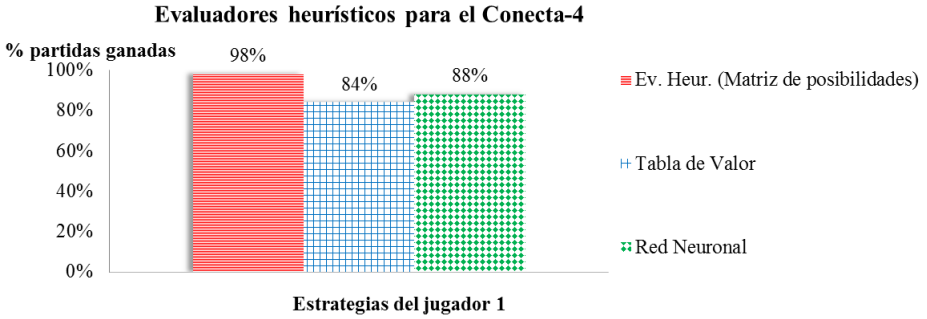


Figura 7.1: Gráfica comparativa de los evaluadores heurísticos del Conecta-4 para el primer jugador.

Tabla 7.2: Comparativa de los evaluadores heurísticos del Conecta-4 para el segundo jugador.

Jugador 2:	**Ev. Heur. (Matriz de posibilidades)**	**Tabla de valor**	**Red neuronal**
Gana	94 %	78 %	81 %
Empata	0 %	0 %	1 %
Pierde	6 %	22 %	18 %

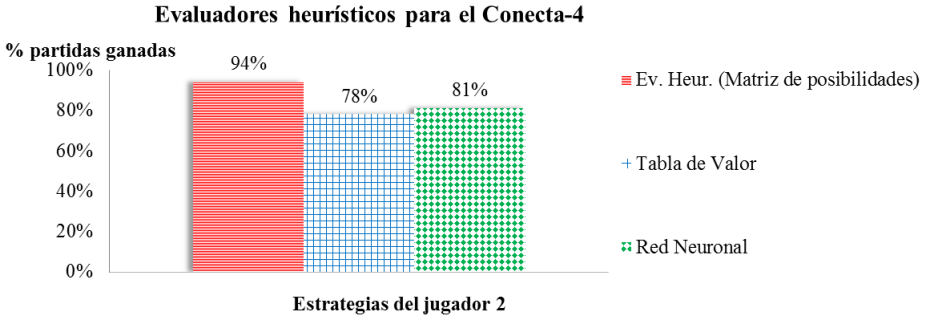


Figura 7.2: Gráfica comparativa de los evaluadores heurísticos del Conecta-4 para el segundo jugador.

Se observa que un evaluador que utilice información de la situación actual del juego para realizar la evaluación ofrece mejores resultados que un evaluador que no tenga en cuenta esa información; aunque en este caso, con un número mayor de partidas de entrenamiento se pueden conseguir los mismos resultados.

7.1.2. Evaluadores del Go

Para el juego del Go también también se han jugado 100 partidas entre un `JugadorEvaluar` y un jugador aleatorio para cada prueba. Las pruebas se han realizado en un tablero de dimensiones 9x9, con las reglas de puntuación japonesas y sin puntos de ventaja para el segundo jugador.

Las tablas 7.3 y 7.4 presentan los resultados de las pruebas para el juego del Go y las figuras 7.3 y 7.4 muestran de forma gráfica los resultados. Los evaluadores heurísticos usados han sido un evaluador basado en el número de territorios (Territorios), uno basado en los puntos según las reglas chinas (P. CH), otro basado en los puntos según las reglas japonesas (P. JP), la tabla de valor y la red neuronal. La tabla de valor y la red neuronal se han entrenado con 2.000 partidas frente a un jugador aleatorio, con los mismos parámetros de entrenamiento que en el experimento anterior. La red neuronal cuenta con nueve neuronas en la capa intermedia.

Tabla 7.3: Comparativa de los evaluadores heurísticos del Go para el primer jugador.

Jugador 1:	**Territorios**	**P. CH**	**P. JP**	**Tabla de valor**	**Red neuronal**
Gana	94 %	100 %	97 %	72 %	99 %
Empata	0 %	0 %	0 %	0 %	0 %
Pierde	6 %	0 %	3 %	24 %	1 %

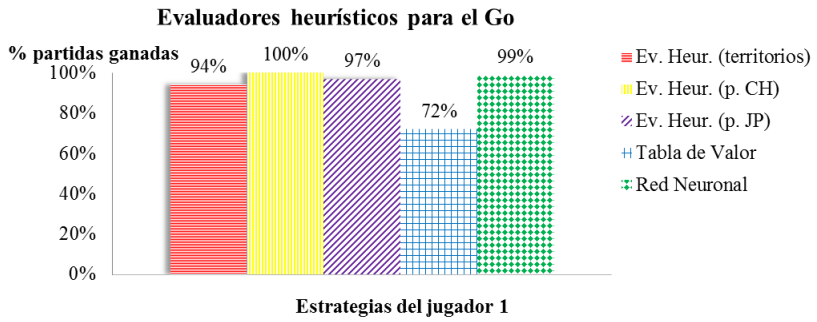


Figura 7.3: Gráfica comparativa de los evaluadores heurísticos del Go para el primer jugador.

En el caso del Go la diferencia entre los evaluadores que usan información del dominio y los que no es más evidente, sobre todo para el segundo jugador; esto se debe a que el tamaño del espacio de estados del Go es mayor que el del Conecta-4, lo que además hace que una estrategia aleatoria sea muy fácil de vencer. Los evaluadores entrenables también necesitan más partidas de entrenamiento para conseguir los mismos resultados que el resto de evaluadores.

Tabla 7.4: Comparativa de los evaluadores heurísticos del Go para el segundo jugador.

Jugador 2:	**Territorios**	**P. CH**	**P. JP**	**Tabla de valor**	**Red neuronal**
Gana	93 %	99 %	95 %	67 %	65 %
Empata	0 %	0 %	0 %	1 %	2 %
Pierde	7 %	1 %	5 %	32 %	33 %

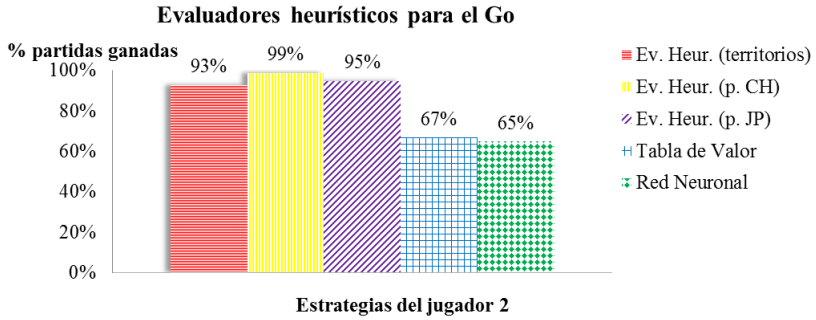


Figura 7.4: Gráfica comparativa de los evaluadores heurísticos del Go para el segundo jugador.

7.1.3. Entrenamientos

En esta sección se estudian varias secuencias de entrenamiento en el juego del Go. Se ha considerado un tablero 9x9 con las reglas de puntuación japonesas y sin ninguna ventaja para el segundo jugador.

Se entrenan cuatro jugadores diferentes con red neuronal (`JugadorEvaluar`): uno como primer jugador frente a una estrategia aleatoria (J1), otro como segundo jugador frente a una estrategia aleatoria (J2) y los otros dos jugadores se entrenan simultáneamente uno frente al otro (J3 y J4). En todos los casos la red neuronal dispone de nueve neuronas en la capa intermedia y los pesos iniciales son aleatorios, la tasa de aprendizaje para ajustar la velocidad a la que aprende la red tiene un valor de 0.1 y el momento de entrenamiento que indica la influencia que tiene la iteración anterior sobre la actual es de 0.8.

Las tablas 7.5, 7.6 y 7.6 muestran cómo evoluciona el aprendizaje de la red neuronal para cada uno de los casos. Las tres secuencias de entrenamiento constan de 2.000 partidas, mostrándose cada 100 los resultados de 100 partidas sin aprendizaje. En todos los casos se toman las últimas redes, es decir, las obtenidas después de los entrenamiento completos de 2.000 partidas, independientemente de los resultados obtenidos durante el proceso. Esto puede dar lugar a un sobreaprendizaje, como se observa en la secuencia 3.

Las figuras 7.5, 7.6 y 7.7 muestran de forma gráfica la evolución del aprendizaje para cada secuencia de entrenamiento.

En el aprendizaje pueden presentarse oscilaciones, por lo que puede ser necesario cierto grado de experimentación antes de obtener una red correctamente entrenada.

Los valores de la secuencia número 3 pueden parecer extraños, esto se debe a que la red neuronal siempre devuelve el mismo movimiento para un estado dado y por lo tanto siempre se juegan las mismas partidas entre las dos redes neuronales; sin embargo, es más provechoso entrenar a dos jugadores simultáneamente (cada uno aprendiendo de su juego con el otro) que usando un jugador aleatorio. A medida que se aumenta el tamaño del tablero de juego, el jugador aleatorio es menos eficaz para entrenar a los jugadores.

Con los jugadores entrenados, cada uno jugará 100 partidas frente a una estrategia aleatoria para comparar los resultados de los entrenamientos. La tabla 7.8 y la figura 7.8 presentan los

Tabla 7.5: Secuencia 1.

Nº partidas	J1	Empate	Aleatorio
0	42 %	4 %	54 %
100	41 %	3 %	56 %
200	24 %	8 %	68 %
300	37 %	3 %	60 %
400	21 %	5 %	74 %
500	25 %	4 %	71 %
600	63 %	3 %	34 %
700	67 %	1 %	32 %
800	43 %	3 %	54 %
900	61 %	0 %	39 %
1000	57 %	0 %	43 %
1100	87 %	0 %	13 %
1200	87 %	0 %	13 %
1300	73 %	0 %	27 %
1400	94 %	0 %	6 %
1500	99 %	0 %	1 %
1600	98 %	0 %	2 %
1700	97 %	0 %	3 %
1800	98 %	0 %	2 %
1900	98 %	0 %	2 %
2000	96 %	0 %	4 %

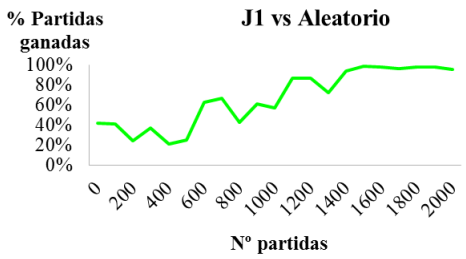


Figura 7.5: Secuencia 1.

Tabla 7.6: Secuencia 2.

Nº partidas	Aleatorio	Empate	J2
0	50 %	0 %	48 %
100	35 %	0 %	61 %
200	43 %	0 %	51 %
300	53 %	0 %	45 %
400	37 %	0 %	61 %
500	35 %	0 %	65 %
600	49 %	0 %	51 %
700	51 %	0 %	47 %
800	8 %	0 %	92 %
900	6 %	0 %	94 %
1000	14 %	0 %	85 %
1100	17 %	0 %	82 %
1200	33 %	0 %	65 %
1300	15 %	0 %	85 %
1400	11 %	0 %	89 %
1500	15 %	0 %	85 %
1600	12 %	0 %	88 %
1700	18 %	0 %	82 %
1800	16 %	0 %	84 %
1900	7 %	0 %	93 %
2000	33 %	0 %	67 %

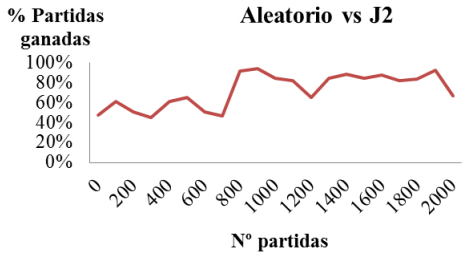


Figura 7.6: Secuencia 2.

resultados de estas pruebas. Se observa que en el caso del jugador J4 se han obtenido mejores resultados que para el jugador J2. Recordemos que J4 entrena frente a otro jugador que aprende simultáneamente (J3), mientras J2 entrena frente a un jugador aleatorio.

7.2. Comparativa de estrategias

En este sección se comparan algunas de las estrategias desarrolladas; para ello se analizan varios estados de los juegos y se realizan algunas simulaciones.

En primer lugar se compara la estrategia minimax frente a su versión mejorada con poda alfa-beta. También se comparan las dos versiones desarrolladas del método de Monte-Carlo. En estos casos se usará el juego del Conecta-4. Por último se realizan varias pruebas en forma de simulaciones, donde cada estrategia juega frente a un jugador aleatorio en el juego del Go.

Tabla 7.7: Secuencia 3.

Nº partidas	Aleatorio	Empate	J2
0	0 %	0 %	100 %
100	0 %	0 %	100 %
200	0 %	0 %	100 %
300	100 %	0 %	0 %
400	100 %	0 %	0 %
500	100 %	0 %	0 %
600	100 %	0 %	0 %
700	100 %	0 %	0 %
800	0 %	0 %	100 %
900	100 %	0 %	0 %
1000	0 %	0 %	100 %
1100	100 %	0 %	0 %
1200	0 %	0 %	100 %
1300	0 %	100 %	0 %
1400	0 %	100 %	0 %
1500	0 %	100 %	0 %
1600	0 %	100 %	0 %
1700	100 %	0 %	0 %
1800	0 %	100 %	0 %
1900	100 %	0 %	0 %
2000	100 %	0 %	0 %

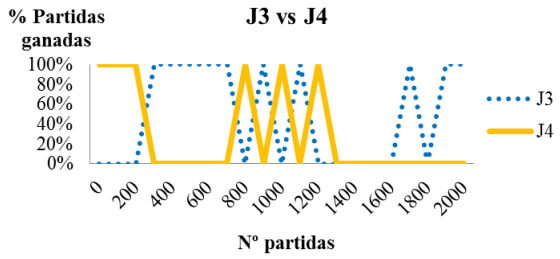


Figura 7.7: Secuencia 3.

7.2.1. Minimax y Alfa-Beta

A continuación se compara la eficacia de un jugador con poda alfa-beta frente a un jugador con el algoritmo minimax en su versión simple. Se estudian dos estados del juego del Conecta-4 en un tablero 6x7: un estado inicial del juego y un estado intermedio en el que ya se han realizado varios movimientos.

Dado el estado inicial del juego del Conecta-4 (figura 7.9) se realiza una búsqueda en el árbol de juegos mediante las estrategias minimax y alfa-beta para obtener el mejor movimiento. En ambos casos se limita la profundidad máxima de búsqueda al nivel 7.

En la tabla 7.9 se muestran las estadísticas sobre la búsqueda del mejor movimiento posible. Para cada nivel (desde 0 hasta la profundidad máxima) se muestra el número de nodos examinados en ese nivel por cada estrategia. También se muestra el tiempo empleado en realizar la búsqueda.

El mejor movimiento devuelto por ambas estrategias se muestra en la figura 7.10, que co-

Tabla 7.8: Resultados frente a una estrategia aleatoria.

Jugador 1:	**J1**	**Aleatorio**	**J3**	**Aleatorio**
Jugador 2:	**Aleatorio**	**J2**	**Aleatorio**	**J4**
Gana jugador 1	99 %	33 %	78 %	7 %
Empate	0 %	2 %	3 %	1 %
Gana jugador 2	1 %	65 %	19 %	92 %

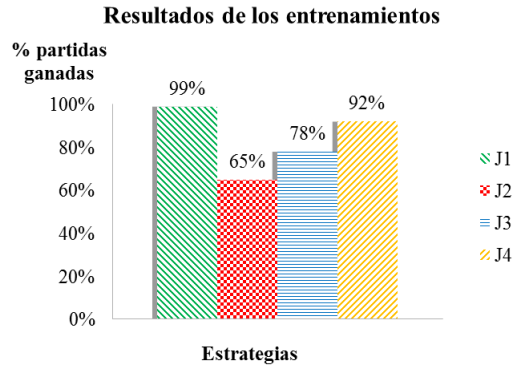


Figura 7.8: Resultados de los entrenamientos.

rresponde a soltar una ficha en la columna central; obviamente tanto minimax como alfa-beta devuelven el mismo movimiento. Se observa el importante ahorro de tiempo usando la poda alfa-beta; en algunos casos se llegan a podar ramas completas del árbol de juegos.

En el siguiente ejemplo se considera un tablero ya inicializado correspondiente al estado mostrado en la figura 7.11. Le toca mover al primer jugador (*MAX*) que juega con *O*. En este caso las estrategias minimax y alfa-beta disponen de un límite de tiempo de un segundo para realizar la búsqueda del mejor movimiento.

La tabla 7.10 presenta las estadísticas obtenidas en esta prueba. Se muestra el número de nodos expandidos en cada iteración, teniendo en cuenta que las estrategias realizan una profundización progresiva.

Mientras que minimax alcanza una profundidad máxima de búsqueda de 6 niveles, alfa-beta consigue llegar hasta una profundidad de 8 niveles en el mismo tiempo. El número de nodos expandidos con la estrategia alfa-beta es sólo orientativo pues en cada profundización pueden

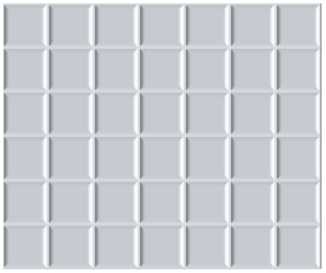


Figura 7.9: Estado inicial del Conecta-4.

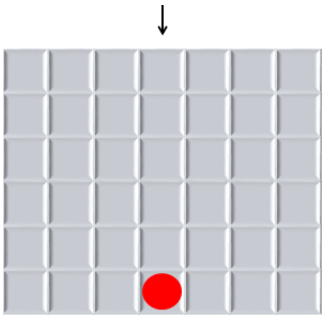


Figura 7.10: Mejor movimiento devuelto por las estrategias minimax y alfa-beta para el estado inicial del Conecta-4.

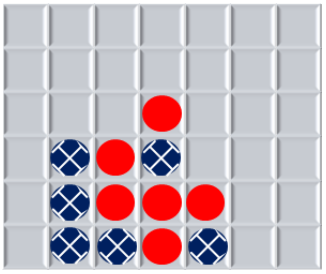


Figura 7.11: Estado inicializado del Conecta-4.

expandirse más o menos nodos dependiendo de si se producen o no podas.

La figura 7.12 muestra el mejor movimiento devuelto por las dos estrategias, que en este caso es también el mismo, aunque ya no tienen por qué coincidir, pues cada estrategia evalúa los estados a diferente profundidad. El estado dado era favorable para el segundo jugador (*MIN*), de ahí que el mejor movimiento posible para *MAX* sea obligado en la segunda columna para no perder la partida. *MIN* juega una partida perfecta y ganaría en la siguiente jugada si *MAX* no moviera en la primera columna.

Tabla 7.9: Estadísticas de las estrategias minimax y alfa-beta.

	Minimax	**Alfa-Beta**
Profundidad	**Nº nodos examinados**	**Nº nodos examinados**
0	1	1
1	7	7
2	49	27
3	343	118
4	2.401	427
5	16.807	1.621
6	117.649	5.908
7	823.536	22.103
Total:	960.793 nodos	30.212 nodos
Tiempo:	12,32 segundos	1,10 segundos

Tabla 7.10: Estadísticas de las estrategias minimax y alfa-beta con límite de tiempo.

	Minimax	**Alfa-Beta**
Iteración	**Nº nodos expandidos**	**Nº nodos expandidos**
0	1	1
1	8	8
2	57	37
3	357	131
4	2326	607
5	14492	1721
6	66420	2184
7		13909
8		57439
Total:	83661 nodos	76037 nodos

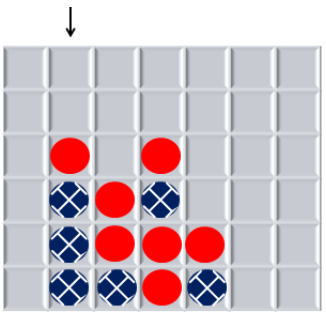


Figura 7.12: Mejor movimiento de minimax y alfa-beta para el estado de la figura 7.11.

7.2.2. Monte-Carlo y Monte-Carlo Tree Search

Este apartado compara las dos estrategias que usan el método de Monte-Carlo. Para ello se estudian los resultados de enfrentar ambas estrategias en función del límite de tiempo disponible para decidir el mejor movimiento.

Se considera el juego del Conecta-4 en un tablero 6x7. Cada prueba consiste en 100 partidas con un límite de tiempo para cada jugador a la hora de decidir los movimientos. Para cada prueba se incrementa el límite de tiempo en un segundo, salvo para la última prueba que tiene un límite de 15 segundos.[1]

En el primer caso, el jugador que mueve en primer lugar usa el método de Monte-Carlo básico y el segundo jugador el método Monte-Carlo Tree Search. La tabla 7.11 muestra los resultados de las pruebas realizadas. La figura 7.13 muestra estos datos de forma gráfica.

En el siguiente caso, el primer jugador usa el método Monte-Carlo Tree Search y el segundo jugador el método básico de Monte-Carlo. Los resultados de esta pruebas se muestran en la tabla 7.12 y en la figura 7.14.

En ambos casos, jugando como primer jugador o como segundo jugador, se observa cla-

[1]Cada una de las pruebas puede durar varias horas, de ahí que se haya escogido el juego del Conecta-4 en lugar del juego del Go para realizar este experimento; incluso en el juego del Conecta-4 en un tablero 6x7 la duración media de cada prueba ha sido de 4 horas.

ramente que el método Monte-Carlo Tree Search obtiene mejores resultados que el método Monte-Carlo basado simplemente en las simulaciones.

Tabla 7.11: Comparativa del método Monte-Carlo frente a Monte-Carlo Tree Search.

Límite de tiempo (s)	Gana jug.1 (MC)	Empate	Gana jug.2 (MC Tree Search)
1	16 %	5 %	79 %
2	12 %	1 %	87 %
3	11 %	1 %	88 %
4	3 %	0 %	97 %
5	3 %	1 %	96 %
6	4 %	2 %	94 %
7	4 %	0 %	96 %
8	3 %	1 %	96 %
9	3 %	1 %	96 %
10	3 %	3 %	94 %
15	1 %	0 %	99 %

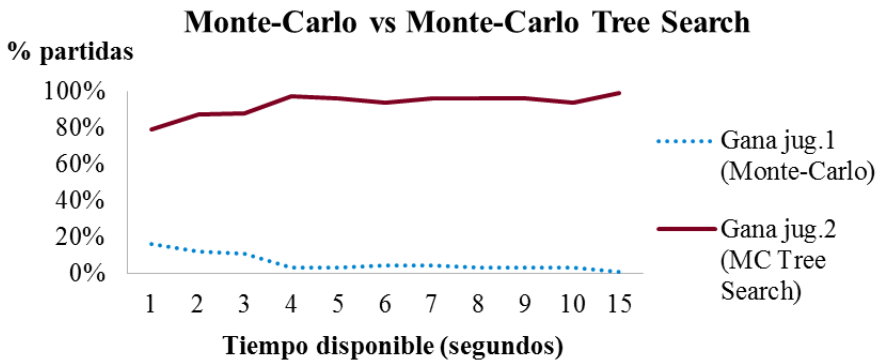


Figura 7.13: Gráfica comparativa de los métodos de Monte-Carlo (datos de la tabla 7.11).

Tabla 7.12: Comparativa del método Monte-Carlo Tree Search frente a Monte-Carlo.

Límite de tiempo (s)	Gana jug.1 (MC Tree Search)	Empate	Gana jug.2 (MC)
1	79 %	3 %	18 %
2	83 %	6 %	11 %
3	90 %	7 %	3 %
4	87 %	6 %	7 %
5	96 %	4 %	0 %
6	94 %	4 %	2 %
7	95 %	2 %	3 %
8	95 %	3 %	2 %
9	98 %	2 %	0 %
10	98 %	1 %	1 %
15	99 %	1 %	0 %

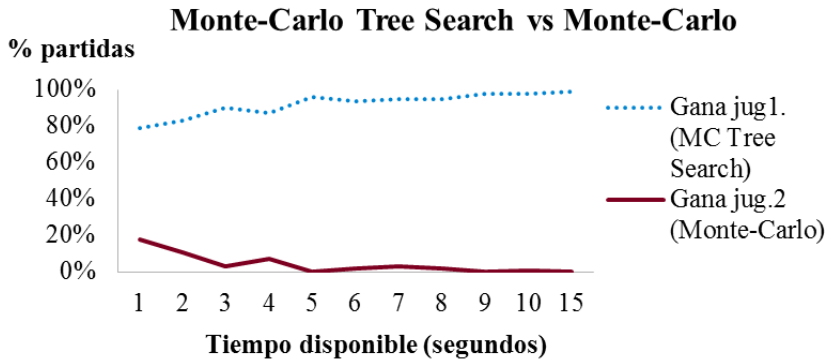


Figura 7.14: Gráfica comparativa de los métodos de Monte-Carlo (datos de la tabla 7.12).

Tabla 7.13: Comparativa de las estrategias usadas por el primer jugador en el Go.

Jugador 1:	**Aleatorio**	**Ev. Heur.**	**Minimax**	**Alfa-Beta**	**MC**	**MC Tree Search**
Gana	47 %	98 %	89 %	96 %	83 %	88 %
Empata	4 %	1 %	0 %	0 %	1 %	2 %
Pierde	49 %	1 %	11 %	4 %	16 %	10 %

7.2.3. Comparativa de estrategias en el Go

A continuación se realiza una comparativa de varias estrategias en el juego del Go.

Se considera un tablero 9x9 inicialmente vacío, con las reglas de puntuación japonesas y sin puntos de ventaja para el segundo jugador. Las estrategias a comparar son un jugador aleatorio, un jugador evaluador heurístico (`JugadorEvaluar`), un jugador minimax, un jugador con poda alfa-beta, un jugador Monte-Carlo y otro Monte-Carlo Tree Search. El heurístico empleado tanto para el jugador evaluador heurístico como para minimax y alfa-beta ha sido el evaluador basado en la puntuación según las reglas japonesas. Todas las estrategias (salvo el `JugadorEvaluar`) disponen de un segundo para encontrar el mejor movimiento.

La tabla 7.13 presenta los resultados de las pruebas para el caso en el que las estrategias juegan como primer jugador. Cada estrategia ha jugado 100 partidas frente a un jugador aleatorio. La figura 7.15 muestra de forma gráfica los resultados obtenidos.

Los resultados de las estrategias cuando son usadas por el segundo jugador se muestran en la tabla 7.14. La figura 7.16 muestra de forma gráfica estos resultados. En este caso también se han jugado 100 partidas con cada estrategia frente a un jugador aleatorio.

El evaluador heurístico (`JugadorEvaluar`) obtiene mejores resultados que minimax y alfa-beta porque examina completamente el primer nivel del árbol de estados. Minimax y alfa-Beta realizan la búsqueda del mejor movimiento primero en profundidad y en un sólo segundo no les da tiempo a examinar el primer nivel completo.

El método de Monte-Carlo básico obtiene mejores resultados en este caso que Monte-Carlo Tree Search. Esto se debe a que las simulaciones de Monte-Carlo Tree Search tardan más en ejecutarse que las simulaciones simples de Monte-Carlo. Un único segundo es muy poco tiempo para realizar las simulaciones de Monte-Carlo Tree Search en un juego como el Go, ya que

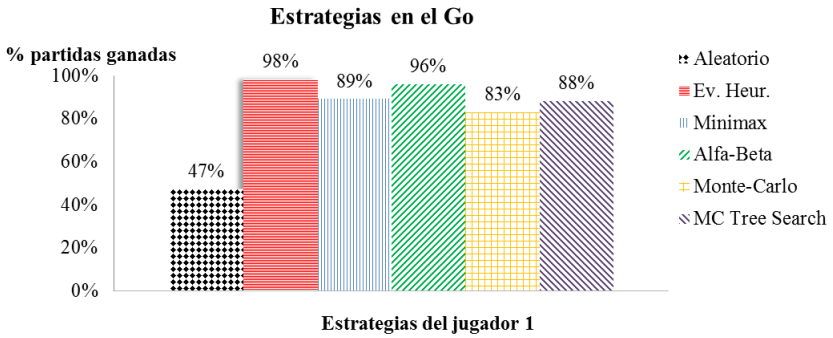


Figura 7.15: Gráfica comparativa de las estrategias del primer jugador en el juego del Go.

Tabla 7.14: Comparativa de las estrategias usadas por el segundo jugador en el Go.

Jugador 2:	**Aleatorio**	**Ev. Heur.**	**Minimax**	**Alfa-Beta**	**MC**	**MC Tree Search**
Gana	49 %	94 %	87 %	89 %	80 %	64 %
Empata	4 %	0 %	1 %	0 %	0 %	2 %
Pierde	47 %	6 %	12 %	11 %	20 %	34 %

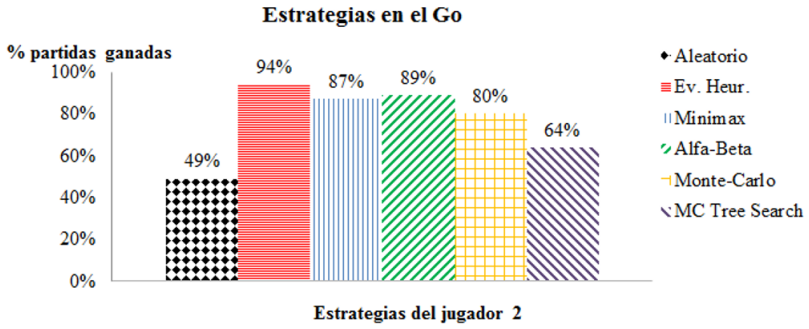


Figura 7.16: Gráfica comparativa de las estrategias del segundo jugador en el juego del Go.

tanto el árbol de búsqueda como el árbol que se genera son enormes. Aumentando el tiempo disponible se mejora esta estrategia igual que ocurría en el experimento de la sección 7.2.2.

Capítulo 8

Conclusiones y trabajo futuro

Este capítulo presenta las conclusiones obtenidas tras el desarrollo del proyecto. También describe el trabajo futuro relacionado con el proyecto como posibles extensiones o aplicaciones en otros campos de la IA.

8.1. Conclusiones

Se han alcanzado plenamente los objetivos del proyecto: desarrollar un entorno interactivo para estudiar las estrategias de IA en los juegos. La aplicación desarrollada permite ver en funcionamiento las estrategias de juegos, pudiendo jugar contra ellas y simplificando la tarea de analizarlas y compararlas. Se trata, por lo tanto, de una herramienta didáctica que puede resultar muy útil de cara a la docencia, tanto para los alumnos como para los profesores.

La búsqueda de una estrategia óptima en juegos es un problema intratable en la mayoría de los casos, todos los algoritmos deben hacer algunas suposiciones y aproximaciones. Los algoritmos vistos son solamente algunas formas de aproximarse a la estrategia óptima. Cada estrategia tiene sus ventajas e inconvenientes, por ejemplo minimax garantiza una estrategia ganadora si pudiera aplicarse al árbol de juegos completo, pero como esto no es posible en la práctica, da lugar a varios inconvenientes como el *efecto horizonte*[1] cuando la condición de corte del algoritmo está basada sólo en una profundidad fija.

Se han desarrollado y estudiado en detalle los algoritmos clásicos en sus versiones más básicas (minimax, alfa-beta y tablas de transposición), dejando de lado posibles mejoras más sofisticadas de estos como la búsqueda de la quietud (*quiescence search*), los números de conspiración (*conspiracy numbers*), la búsqueda con ventana cero (*zero window search*) o las bases de datos de aperturas y de finales,[2] aunque el proyecto permite agregar estas estrategias de forma sencilla.

También se han estudiado otros métodos más recientes como Monte-Carlo Tree Search,

[1] Efecto que ocurre cuando se evalúa como buena o mala una posición sin saber que en la siguiente jugada la situación se revierte.

[2] Puede obtenerse más información sobre estos conceptos en [HA10].

basado en el clásico método de Monte-Carlo. A su vez, se han visto una rama de la IA como es el aprendizaje en máquinas, aplicado a los agentes de juegos; permitiendo entrenar redes neuronales capaces de aprender a jugar al Conecta-4 y al Go.

Como curiosidad mencionar una pequeña experiencia que tuve durante el proyecto en mi búsqueda de empleo: una empresa británica relacionada con las apuestas deportivas por Internet me propuso como ejercicio de evaluación el desarrollo del juego del 3 en raya junto con un jugador inteligente, usando el lenguaje Java. No había ninguna restricción en cuanto a la estrategia del jugador, pero la única condición que debía cumplir es que nunca perdiera frente a la estrategia propuesta por ellos. La estrategia Monte-Carlo en su versión básica con 1.000 simulaciones por movimiento fue suficiente para cumplir el objetivo.

8.2. Trabajo futuro

Como trabajo futuro se presentan algunas extensiones y mejoras del proyecto realizado que pueden dar lugar a otros proyectos de investigación en el ámbito de la IA y la ingeniería del software.

8.2.1. Extensiones del proyecto

A parte de la extensión natural del proyecto incorporando nuevos juegos, estrategias y heurísticos como se muestra en el apéndice B, el proyecto queda abierto a otro tipo de extensiones y mejoras de mayor envergadura. A continuación se exponen algunas de ellas ordenadas de mayor a menor relevancia:

- Adaptación del módulo de razonamiento a otras clases de juegos: multijugador, de suma no cero, de información imperfecta e indeterministas.

  Aunque el proyecto esta enfocado a los juegos clásicos de tablero, la extensión más natural del mismo es extender su aplicación a otros tipos de juegos con diferentes características.

  Esta extensión requiere rediseñar el módulo de razonamiento además de adaptar todas las estrategias a las nuevas características de los juegos; teniendo en cuenta que no todos los algoritmos vistos son generalizables para las nuevas características. Esta extensión puede considerarse un proyecto aparte debido a su alcance y complejidad.

- Mejoras de los algoritmos con versiones concurrentes de los mismos.

  Con la consolidación de la tecnología Grid y los procesadores paralelos se dispone de mayores recursos (cómputo y almacenamiento) para mejorar las decisiones de los algoritmos clásicos; además de desarrollar nuevos algoritmos basados en la programación distribuida.

- Modificación de la aplicación interactiva para incorporar nuevos juegos, estrategias y heurísticos en tiempo de ejecución.

  Esta modificación, aunque pueda parecer una mejora obvia y deseada en la aplicación, supone una tarea compleja de llevar a cabo, debido a que se debe dividir las diferentes partes del módulo de razonamiento de forma que puedan compilarse por separado y después conectarse a la aplicación. Se trata de un desarrollo basado en componentes.

  Dentro de esta extensión también puede considerarse la posibilidad de que los nuevos componentes sean desarrollados con diferente tecnología o que sean independientes del lenguaje de programación.

Personalmente la primera extensión es la más atractiva, pues expande el ámbito de los juegos con las nuevas características de los mismos, sobre todo con el auge actual de los juegos modernos de tablero (*modern board-games* o *Eurogames*); esto obliga a estudiar e investigar nuevas estrategias o modificar las existentes para ajustarse a dichas características.

8.2.2. Aplicación a otras áreas

Los juegos se han representado como problemas de búsqueda con adversarios y los algoritmos estudiados están enfocados a ese tipo de problemas. Sin embargo, estos algoritmos se pueden extender a otras áreas de la Inteligencia Artificial o de la Investigación Operativa como por ejemplo la generación de planes o los problemas de decisión.

También ocurre el proceso inverso, es decir, que se adapten algoritmos de otras áreas al ámbito de los juegos, como es el caso del método de Monte-Carlo, usado en infinidad de campos (economía, finanzas [Gla04], física médica, procesamiento de gráficos [SlB10] o incluso para calcular el número *pi*). En [Fis96] puede encontrarse más aplicaciones del método de Monte-Carlo.

En ambos casos, las continuas innovaciones en el ámbito de los juegos generan entusiasmo y resultan relevantes para las investigaciones en IA.

Apéndices

Apéndice A

Manual de usuario

Este apéndice es el manual de usuario del entorno interactivo. Muestra la instalación de la aplicación y explica el funcionamiento de la misma.

A.1. Instalación

El entorno interactivo no necesita de un proceso de instalación. Se trata de un archivo *JAR* (*Java Archive*), lo que facilita la distribución de la aplicación.

A.1.1. Requisitos mínimos

El requisito mínimo para la ejecución de la aplicación es:

- Tener instalada la Máquina Virtual de Java (JVM) en su versión 1.6 o superior.
  Se puede obtener gratuitamente en: `http://www.java.com/es/download/index.jsp`

A.2. Ejecución

La aplicación se puede ejecutar simplemente haciendo doble clic sobre el fichero *JAR*. También es posible ejecutarlo desde un terminal de línea de comandos con la siguiente instrucción: `java -jar «aplicación.jar»`

A.3. Funcionamiento

Tanto el aspecto de la aplicación como su funcionamiento son fáciles de entender. Todas la ventanas de la aplicación contienen en su parte inferior una zona dedicada a mostrar información de ayuda sobre el componente de la ventana en el que se sitúa el usuario con el puntero del ratón.

El programa dispone de una pantalla principal dividida en tres partes: los juegos, las estrategias y las funciones principales. La figura A.1 muestra la ventana principal.

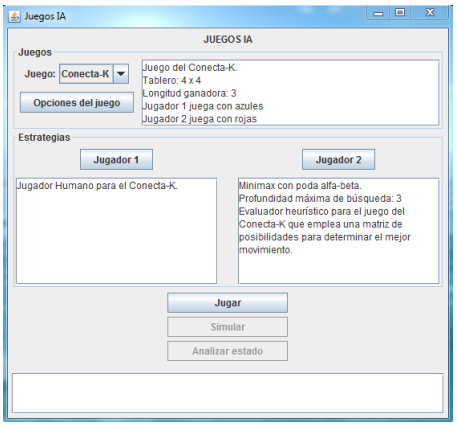


Figura A.1: Pantalla principal de la aplicación.

La parte de los juegos nos permite seleccionar y configurar un juego; la parte de estrategias se encarga de seleccionar, configurar y entrenar si es necesario las estrategias de los dos jugadores y por último tenemos los tres modos de uso de la aplicación: jugar, simular y analizar estado.

Estas partes están organizadas de arriba a abajo, pues el orden de uso es importante: primero se debe elegir y configurar el juego, a continuación las estrategias de los dos jugadores y por último elegir la acción deseada. Esto es así porque las estrategias pueden necesitar información del juego y este último debe estar configurado previamente. Si seleccionamos un juego después de configurar las estrategias, las estrategias se restablecerán a sus valores por defecto.

A.3.1. Selección y configuración del juego

Se puede seleccionar un juego directamente desde la pantalla principal de la aplicación; junto a él se muestra información del juego y su configuración por defecto.

Si se desea cambiar la configuración del juego debemos seleccionar la opción ‘‘`Opciones del juego`’’. Se accede a una nueva ventana con diferentes opciones de configuración que dependen del juego seleccionado. Para obtener ayuda sobre estas opciones hay que situar el puntero del ratón sobre las mismas y se obtendrá información sobre ella en la parte inferior de la ventana.

A.3.2. Selección y configuración de los jugadores

La parte central de la ventana principal esta dividida a su vez en dos partes: una dedicada al primer jugador y otra al segundo jugador. Cada una de ellas muestra información sobre la estrategia actual seleccionada para el jugador. La estrategia por defecto es un jugador humano. Pulsando los botones ‘‘`Jugador 1`’’ y ‘‘`Jugador 2`’’ podemos cambiar y configurar las estrategias para cada jugador.

La figura A.2 muestra la ventana de configuración de las estrategias; esta ventana es idéntica para ambos jugadores. Contiene una lista con las estrategias disponibles; al seleccionar una

estrategia se muestra al lado información sobre la misma y la parte inferior de la ventana cambia para mostrar las opciones de configuración de esa estrategia.[1]

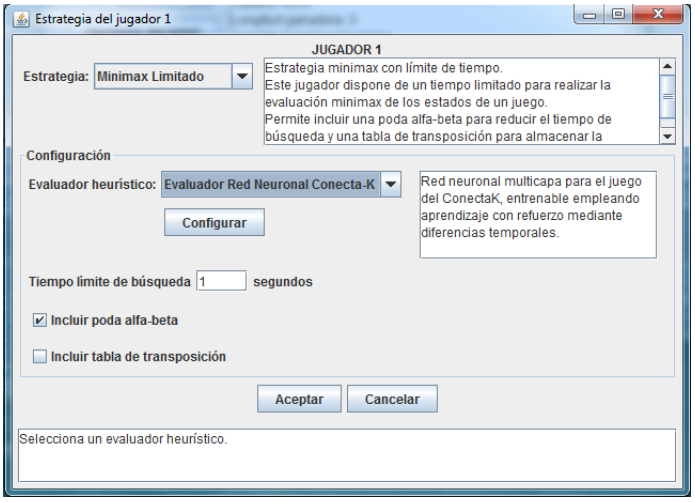


Figura A.2: Pantalla de selección y configuración de la estrategia.

La opción más interesante es la selección y configuración de un evaluador heurístico para las estrategias. Esta opción sólo estará disponible en las estrategias que necesiten de un heurístico para evaluar las posiciones de los juegos. Por otro lado también hay heurísticos que no necesitan configurar sus parámetros.

En el caso de que el heurístico permita configurar sus parámetros se activa la opción ‘‘`Configurar`’’ debajo del heurístico seleccionado. Esta opción da acceso a una nueva ventana como la que se muestra en la figura A.3.

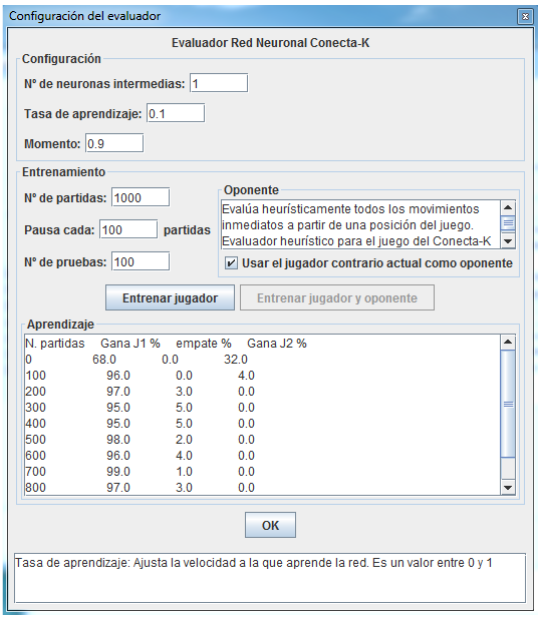


Figura A.3: Pantalla de configuración y entrenamiento de un evaluador heurístico.

[1]No todas las estrategias son configurables.

La parte superior muestra los parámetros de configuración del evaluador heurístico, mientras que la parte inferior permite entrenar el evaluador. La parte de entrenamiento sólo se mostrará si el evaluador es entrenable; en caso contrario la ventana sólo mostrará los parámetros de configuración del evaluador.

Entrenamiento del jugador

La ventana de configuración del evaluador permite entrenar al evaluador heurístico y por tanto al jugador usando aprendizaje por refuerzo.

La parte inferior de la ventana (figura A.3) muestra las opciones de entrenamiento. Para entrenar al jugador se debe indicar el número de partidas de entrenamiento. Los campos ‘‘`pausa cada` n `partidas`’’ y ‘‘`número de pruebas`’’ permiten ver el desarrollo del aprendizaje. Cada n partidas de entrenamiento se realizará una pausa y se jugará el número de partidas de prueba indicado; el resultado de estas pruebas se irá mostrando en la parte inferior.

El entrenamiento se realiza por defecto frente a un oponente con estrategia aleatoria; pero también se puede entrenar frente al jugador contrario activando la opción ‘‘`Usar el jugador contrario actual como oponente`’’. Esta opción sólo se podrá activar si el jugador contrario es controlador por el ordenador, es decir, no es un jugador humano.

La misma ventana de entrenamiento también permite entrenar simultáneamente a los dos jugadores seleccionados. Para ello el jugador contrario debe ser entrenable y debe activarse la opción ‘‘`Usar el jugador contrario actual como oponente`’’ comentada anteriormente. Al activar esta opción aparece un nuevo botón ‘‘`Entrenar jugador y oponente`’’ que sólo estará activo si el jugador oponente también es entrenable. Se entrenará simultáneamente a los dos jugadores, uno frente al otro, cada uno con su propia estrategia.

A.3.3. Modos de uso de la aplicación

La aplicación tiene tres modos de uso básicos: jugar, simular y analizar estado. A continuación se explica el funcionamiento de cada uno de estos modos.

Jugar

Esta opción nos permite jugar una partida al juego seleccionado. Se puede jugar contra cualquier estrategia disponible o contra otro jugador humano. En el caso de que ninguna de las estrategias sea un jugador humano, se mostrará el desarrollo de una partida entre los dos jugadores controlados por el ordenador.

La figura A.4 muestra el desarrollo de una partida de Go entre dos jugadores controlados por el ordenador. La parte derecha muestra información sobre las estrategias y sobre el desarrollo de la partida (último movimiento, turno del jugador, puntuación,...). La parte inferior muestra información sobre el movimiento actual, por ejemplo indicando si estamos sobre una posición sobre la que está prohibido mover.

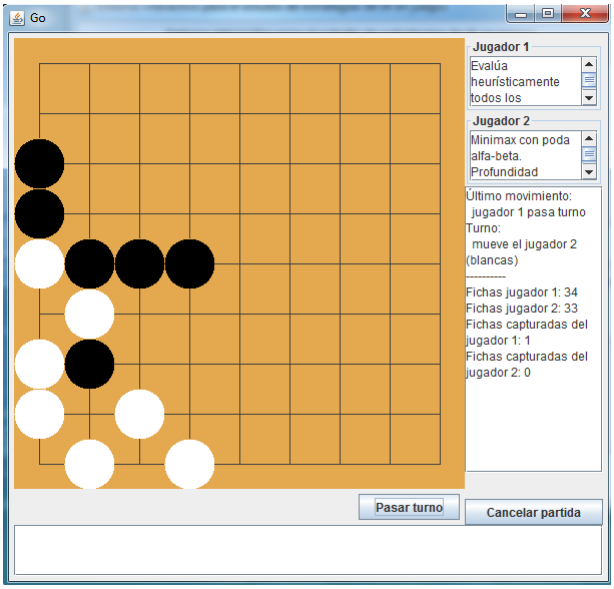


Figura A.4: Pantalla de juego.

Dependiendo de la estrategia usada, los movimientos de los jugadores pueden ser muy rápidos y el ojo humano no es capaz de seguir el desarrollo de la partida; para evitar esto se realiza una pequeña pausa de un segundo entre cada movimiento. La partida puede cancelarse en cualquier momento, volviendo a la pantalla principal.

Cuando termina la partida se muestra la ventana de estadísticas con información acerca de la partida.

Simular

La opción “Simular” consiste en realizar un número determinado de partidas entre dos jugadores controlados por el ordenador. El desarrollo de estas partidas no es visible, pues el propósito de la simulación es comparar las estrategias con los resultados obtenidos a partir de todas las partidas jugadas. La figura A.5 muestra la ventana de simulación.

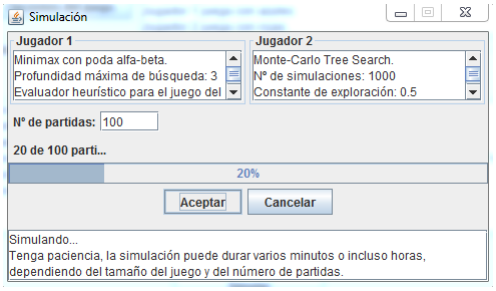


Figura A.5: Pantalla de simulación.

Para realizar una simulación con éxito hay que tener en cuenta el tamaño del árbol de búsqueda del juego seleccionado y las estrategias elegidas porque una simulación puede durar desde segundos hasta horas, en función de estos factores y el número de partidas que se jueguen. La simulación puede cancelarse en cualquier momento.

Al finalizar la simulación se muestran las estadísticas de las partidas en una nueva ventana.

Analizar estado

Esta opción permite al usuario construir un estado del juego seleccionado y a continuación estudiar el movimiento que realizaría cada uno de los jugadores en esa posición.

Para construir el estado el usuario debe realizar los movimientos oportunos de los dos jugadores, situando fichas en las posiciones deseadas pero siempre siguiendo las reglas del juego. El usuario alternará los movimientos de ambos jugadores hasta conseguir el estado deseado.

La figura A.6 muestra la construcción de un estado para el juego del Conecta-4. Una vez construido el estado deseado se debe pulsar la opción ‘‘`Analizar estado`’’ y los dos jugadores elegidos realizarán un movimiento; independientemente del turno del jugador, ambos jugadores realizan el mismo movimiento para comparar la decisión que toman las estrategias usadas por los jugadores.

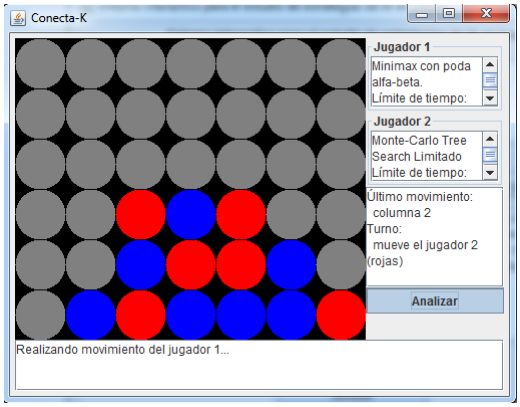


Figura A.6: Pantalla de análisis de estados.

Una vez finalizado el movimiento de los dos jugadores, se mostrará una nueva ventana con información de los movimientos realizados y estadísticas de las estrategias.

A.3.4. Estadísticas

Al finalizar cada uno de los modos de uso de la aplicación se mostrará una ventana con estadísticas. La información mostrada en esta ventana depende del modo de uso elegido, aunque hay información común que es independiente del modo de uso y del jugador como el número de movimientos realizados o el tiempo medio de cada uno de los movimientos.

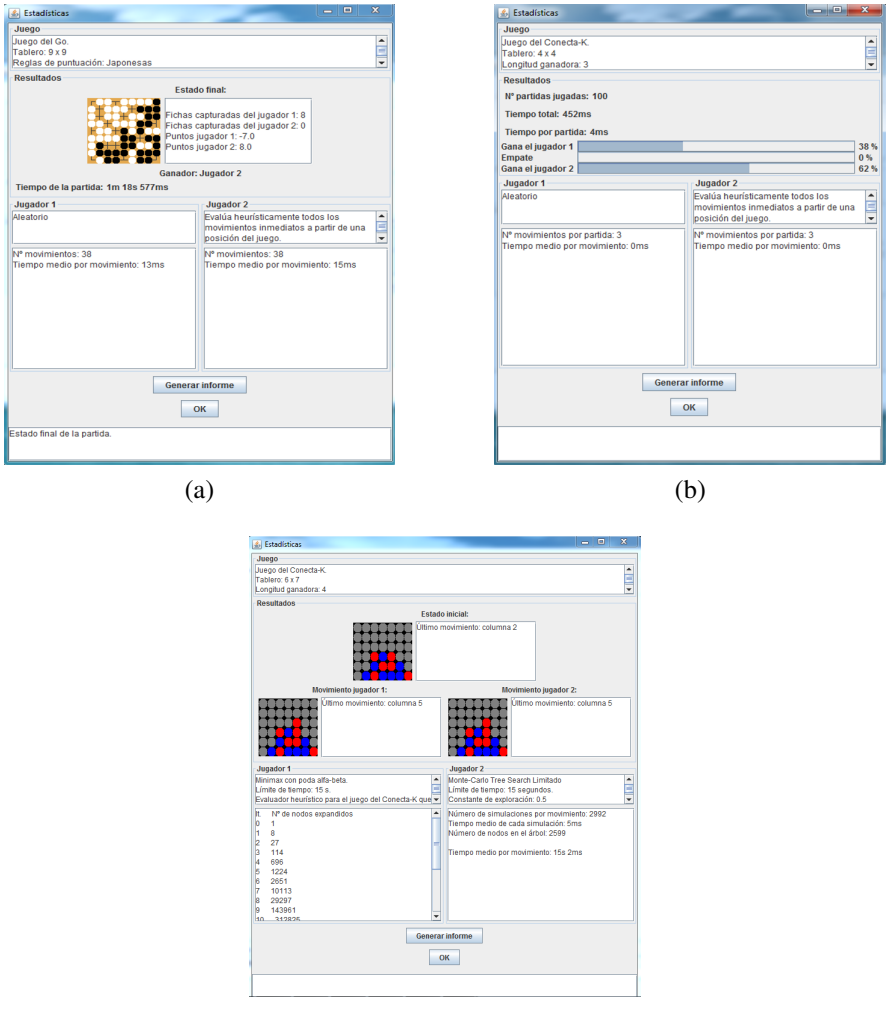
A continuación se describe brevemente la información mostrada para cada modo de uso de la aplicación:

Jugar (Figura A.7a) Al terminar de jugar una partida se muestra la situación del estado final, el último movimiento realizado, el resultado (el ganador), el tiempo de juego[2],... Los jugadores muestran estadísticas de sus estrategias para la partida completa.

[2]El tiempo de juego incluye un segundo adicional por cada movimiento como se comentó en el apartado A.3.3.

Simular (Figura A.7b) En el caso de una simulación se muestra el porcentaje de partidas ganadas por cada jugador, el tiempo total de la simulación y el tiempo medio de cada partida. Hay que prestar atención a la información mostrada por cada uno de los jugadores pues se trata de estadísticas acumuladas de todas las partidas.

Analizar estado (Figura A.7c) Cuando termina el análisis se muestra el estado que ha sido analizado y los estados resultantes después de que los jugadores hayan movido. Cada estado puede mostrar información propia del mismo, aunque esta información depende del juego (último movimiento, puntuación actual de cada jugador,...). Las estadísticas proporcionadas por los jugadores corresponden esta vez a un único movimiento.



(a) (b)

(c)

Figura A.7: Pantallas de resultados y estadísticas.

En todos los casos se puede generar un informe en formato texto plano con toda la información obtenida. Para ello se debe seleccionar la opción "`Generar informe`".

Apéndice B

Desarrollo de nuevos juegos, estrategias y heurísticos

Este apéndice contiene información sobre cómo extender la aplicación, incorporando nuevos juegos, estrategias y heurísticos. Se explica detalladamente como agregar nuevos elementos tanto al módulo de razonamiento como a la interfaz gráfica a partir del diseño propuesto en el capítulo 6.

Para llevar a cabo todas las extensiones que se proponen a continuación es necesario disponer del código fuente del proyecto. Todo el código se encuentra en lenguaje Java (versión 1.6) y las únicas bibliotecas de clases externas que son necesarias son las pertenecientes a la implementación de las redes neuronales (*Encog Machine Learning Framework*); estas se incluyen también en el directorio `lib` dentro del proyecto. En [ENC08] puede obtenerse información sobre dicha biblioteca de clases.

B.1. Desarrollo de juegos

Comenzaremos mostrando como agregar nuevos juegos. Recordemos que los juegos deberán ser bipersonales, por turnos, de suma cero, de información perfecta y deterministas.

Se detallará primero cómo agregar un nuevo juego al módulo de razonamiento y posteriormente a la aplicación gráfica.

B.1.1. Extensión del módulo de juegos

Teniendo en cuenta el diagrama de clases de la figura 6.1 presentado en el capítulo 6, toda clase que represente los estados de un juego debe implementar la interfaz *`EstadoJuego`*; lo que significa que debemos representar los estados de nuestro juego con los siguientes métodos:

`hijos`
: Devuelve una lista con los estados sucesores, es decir, los estados directamente accesibles desde el actual.

`ganador`
: Devuelve la ficha del jugador que ha ganado o *null* si nadie ganó.

`agotado`
: Verdadero si el estado es final y hay empate, falso en otro caso.

`jug1?`
: Verdadero si es el turno del primer jugador, falso en otro caso.

`clave`
: Clave para indexar el estado en formato cadena de texto.

`toString`
: Descripción del estado, como por ejemplo la situación del tablero, turno del jugador, etc.

Para ayudar a implementar esta clase se pueden utilizar las utilidades disponibles en el paquete `Util`. Aunque no es obligatorio su uso, si es recomendable pues proporciona una implementación básica de las fichas, un tablero genérico o una posible representación de los movimientos.

Con esto ya tenemos un nuevo juego en el módulo al que poder aplicarle las estrategias disponibles. Sin embargo, también necesitamos desarrollar un jugador humano que pida los movimientos mediante un dispositivo de entrada. Aunque este jugador no es obligatorio para el módulo de razonamiento, si lo será para el entorno gráfico, por lo que se explica en este apartado.

Jugador humano

Para crear un jugador humano podemos extender la clase abstracta *JugadorHumano* del paquete `Estrategias` o simplemente implementar la interfaz *Jugador* del mismo paquete. En este último caso sólo hay un método que desarrollar:

`mueve`
: Dado el estado actual devuelve el estado resultante de mover en la posición indicada.

El dispositivo de entrada para los movimientos de los juegos en el módulo de razonamiento será normalmente el teclado, aunque para la interfaz gráfica lo normal será usar el ratón y el movimiento será proporcionado por el controlador de la propia interfaz del juego.

A continuación, para añadir el juego al entorno gráfico hay que desarrollar la interfaz gráfica del juego.

B.1.2. Extensión de la aplicación para juegos

Para incorporar el juego a la aplicación hay que proporcionar un panel para configurar las opciones del juego (si fuera necesario), un panel de juego (tablero) y un controlador para guiar el desarrollo de la partida.

El paquete `Extensión Juego` del diagrama presentado en la figura 6.4 muestra estos aspectos. Se debe implementar la interfaz *Juego* de dicho paquete que consta de los métodos:

`nombre`
: Nombre del juego.

`estadoJuego`
: Devuelve el estado inicial del juego con la configuración por defecto.

`información`
: Dado un estado proporciona información básica sobre el mismo como por ejemplo el tamaño del tablero. Esta información se mostrará en la ventana principal de la aplicación cuando se seleccione el juego.

`informaciónDetalle`
: Dado un estado proporciona información detallada sobre el mismo (último movimiento, puntuación,...). Esta información aparecerá en la ventana de estadísticas al final de la partida.

`estrategiaHumana`
: Jugador humano para este juego.

`panelOpciones`
: Proporciona un panel (*PanelOpcionesJuego*) para configurar las opciones del juego; si no es necesario configurar nada devolverá *null*.

`panelJuego`
: Proporciona un panel a modo de tablero gráfico donde se desarrollará la partida (clase *PanelJuego*). Incluye una representación gráfica de los elementos que intervengan en la partida, por ejemplo las fichas.

`controladorJuego`
: Proporciona el controlador del juego (clase *ControladorJuego*) que interactúa directamente con el panel de juego y controla el desarrollo de la partida.

El paquete contiene las tres clases abstractas que se deberán especializar y que son las que devuelven los tres últimos métodos presentados anteriormente. Estas clases son:

- `PanelOpcionesJuego`
  que define los métodos:

`estadoJuego`
: Devuelve un estado inicial del juego configurado con las opciones elegidas en el panel.

`registrarControlador`
: Registra un controlador para el panel (opcional).

- `PanelJuego`
  con los métodos:

  `panelJuegoEstado`
  : Dado un estado del juego devuelve la representación del mismo de manera gráfica a través de un panel.

  `registrarControlador`
  : Registra el controlador de juego en los componentes interactivos del panel.

- `ControladorJuego`
  cuyos principales métodos son:

  `jugar`
  : Este el único método abstracto de esta clase y es llamado al iniciar una partida. Se encarga de controlar el desarrollo de la misma.

  `finJuego`
  : Es el método que se deberá llamar al terminar la partida, indicando el valor de utilidad y un mensaje informativo.

Una vez completado el desarrollo completo del juego se deberá indicar en un fichero de configuración representado mediante la clase `Juegos`, agregando una nueva línea con un identificador y una instancia de la clase que implementa la interfaz *`Juego`*.

La siguiente sección describe la forma de incorporar nuevas estrategias, incluyendo una estrategia humana para los juegos.

B.2. Desarrollo de estrategias

A continuación se muestra la incorporación de nuevas estrategias al sistema. Al igual que con los juegos, primero extenderemos el módulo de estrategias y después la aplicación gráfica.

B.2.1. Extensión del módulo de estrategias

Toda clase que represente un agente jugador debe implementar la interfaz *`Jugador`* del paquete `Estrategias` (figura 6.2). El método ha desarrollar es el mismo que para la estrategia humana presentada en B.1.1:

`mueve`

Devuelve el estado resultante de que el jugador mueva en el estado actual.

Para los jugadores que necesiten de un heurístico puede extenderse la clase `JugadorEvaluar` cuyo constructor recibe el evaluador heurístico a usar y dispone de un método ya implementado para evaluar los estados:

`evalua`

Dado un estado y las fichas de los jugadores devuelve infinito si el estado es final y ganador para *MAX*, infinito negativo si es final y perdedor para *MAX* y el valor de la función heurística proporcionada por el evaluador en otro caso.

Para obtener información detallada sobre el rendimiento de la estrategia, la estrategia deberá proporcionar sus propias estadísticas.

Estadísticas

La estrategia debe proporcionar estadísticas básicas de su uso para poder compararlas con las demás. Para ello deberá implementar la interfaz *`Estadísticas`*:

`numTotalMovimiento`

Número total de movimientos realizados por el jugador. Este número es independiente del número de partidas jugadas por el jugador. Es el número de veces que se ha llamado al método mueve.

`tiempoMedioPorMovimiento`

Tiempo medio por movimiento en milisegundos.[1]

`getEstadisticas`

Proporciona estadísticas propias de la estrategia en formato texto.

`inicializarEstadísticas`

Inicializa las estadísticas del jugador.

B.2.2. Extensión de la aplicación para estrategias

Incorporar la nueva estrategia desarrollada a la aplicación gráfica es más fácil que incorporar un nuevo juego, pues sólo se necesita proporcionar un panel para configurar la estrategia en caso de que necesite configuración.

El paquete `Extensión Estrategia` (figura 6.4) contiene la interfaz *`Estrategia`* que se debe implementar. Los métodos son:

[1] A pesar de que el número de movimientos y el tiempo medio por movimiento pueda obtenerse de forma independiente a la estrategia; se exigen estos datos porque la propia estrategia puede calcular estos valores de forma más precisa.

`nombre`
: Nombre de la estrategia.

`información`
: Proporciona una descripción de la estrategia que se mostrará al seleccionarla en la aplicación.

`estrategia`
: Devuelve la estrategia (instancia de *Jugador*) configurada con las opciones por defecto. Se proporciona el estado del juego seleccionado para el caso en el que la estrategia use un evaluador que necesite información del juego para su configuración.

`panelConfiguración`
: Proporciona el panel de configuración de la estrategia, es decir, una instancia de la clase *PanelConfiguraciónEstrategia* o *null* si la estrategia no necesita ser configurada.

El panel de configuración extenderá la clase *PanelConfiguraciónEstrategia* que contiene el método:

`estrategia`
: Devuelve la estrategia configurada con las opciones indicadas en el panel.

En el caso de necesitar un heurístico para la estrategia se puede utilizar a su vez el panel de selección del evaluador heurístico (clase `PanelSelecciónEvaluador`) que se incluye en módulo de la interfaz y que permite seleccionar un evaluador de entre los disponibles en la aplicación. Este panel filtra los heurísticos por el juego seleccionado.

Por último, al igual que ocurría en el caso de los juegos, es necesario especificar la nueva estrategia en un fichero de configuración, representado esta vez mediante la clase `Estrategias`. Se debe agregar una nueva línea con un identificador para la estrategia y una instancia de la clase que implementa la interfaz *Estrategia*.

La última sección muestra cómo incorporar nuevos heurísticos.

B.3. Desarrollo de heurísticos

El desarrollo de un evaluador heurístico puede realizarse de manera independiente al juego, es decir, sin usar información del estado del juego; aunque en la mayoría de los casos el heurístico necesitará de esa información. La forma de incorporar un nuevo heurístico al módulo de evaluadores es independiente de este aspecto.

B.3.1. Extensión del módulo de evaluadores heurísticos

Para definir un objeto evaluador heurístico se debe implementar la interfaz *Evaluador* del paquete Heurísticos (figura 6.3). Esta interfaz contiene únicamente un método:

evaluación
: Dado un estado y las fichas fichaMAX y fichaMIN de los jugadores, devuelve la evaluación del estado suponiendo que *MAX* juega con fichaMAX y *MIN* con fichaMIN. La evaluación será un valor positivo cuando el estado sea favorable para *MAX*, negativo cuando sea desfavorable y cero cuando sea indiferente.

Evaluadores entrenables

Si queremos que el evaluador pueda ser entrenado mediante aprendizaje con refuerzo debemos implementar además la interfaz *DiferenciasTemporalesDT* disponible en el mismo paquete Heurísticos. La interfaz consta de los siguientes métodos:

estadoGanador
: Enseña al evaluador que el estado dado es un estado ganador.

estadoPerdedor
: Enseña al evaluador que el estado dado es un estado perdedor.

estadoEmpate
: Enseña al evaluador que el estado dado es un estado final de empate.

actualizaDT
: Dado los estados e y e2, entrena al evaluador empleando diferencias temporales y siendo e2 sucesor de e.

Para desarrollar un evaluador con red neuronal para un juego concreto, solamente debemos extender la clase *EvaluadorRedNeuronal* e implementar el método codifica:

codifica
: Dado un estado, devuelve la codificación del mismo empleando el número de neuronas de entrada que se necesiten.

B.3.2. Extensión de la aplicación para heurísticos

Una vez desarrollado el evaluador heurístico, hay que integrarlo a la aplicación gráfica para que pueda ser seleccionado por alguna de las estrategias que usen heurísticos.

El paquete Extensión Evaluador (figura 6.4) contiene la interfaz *Heurístico* que se debe implementar. Los métodos son:

`nombre`
: Nombre del evaluador heurístico.

`información`
: Proporciona una descripción del evaluador. Esta información se mostrará al seleccionarlo en la aplicación.

`entrenable`
: Indica si el evaluador es entrenable o no.

`claseEstadoJuego`
: Indica la subclase de *`EstadoJuego`* de la que depende el evaluador heurístico. Si el evaluador es independiente del tipo de juego debe devolver *null*.

`evaluador`
: Devuelve una instancia del evaluador con los parámetros por defecto. Se proporciona el estado del juego para el caso en el que el evaluador necesite información del juego.

`panelConfiguración`
: Proporciona el panel de configuración del evaluador heurístico, es decir, una instancia de la clase *`PanelConfiguraciónEvaluador`* o *null* si el evaluador no necesita ser configurado.

La clase abstracta *`PanelConfiguraciónEvaluador`* deberá extenderse si se desea proporcionar un panel de configuración para el evaluador; sus métodos son:

`evaluador`
: Devuelve una instancia del evaluador configurado con los parámetros indicados en el panel.

`registrarControlador`
: Registra un controlador para el panel (opcional).

También hay que especificar el nuevo heurístico en un fichero de configuración (al igual que ocurre con los juegos y las estrategias); en este caso el fichero de configuración es representado por la clase `Heurísticos`. Se debe agregar una nueva línea con un identificador para el evaluador y una instancia de la clase que implementa la interfaz *`Heurístico`*.

Todos los paneles presentados en las secciones anteriores para cada elemento (juegos, estrategias y heurísticos) pueden proporcionar información de ayuda al usuario mediante el sistema de ayuda del entorno interactivo.

B.4. Sistema de ayuda

Para proporcionar información de ayuda al usuario todos las ventanas y paneles de la aplicación implementan la interfaz *InformaciónAyuda* (figura 6.4):

`infoComponente`
: Devuelve la información de ayuda perteneciente al componente indicado.

`actualizarInfo`
: Actualiza la información del componente activo en este momento. Sólo es necesario si la ventana se encarga de mostrar la información ella misma.

`registrarControladorInformacion`
: Registra en cada componente que desee proporcionar información un controlador específico para el sistema de presentación de la ayuda.

La información de ayuda siempre se muestra en la ventana activa en cada momento, por lo que los paneles internos sólo deben proporcionar la información.

Referencias

[CBSS08] Guillaume Chaslot, Sander Bakkes, Istvan Szita, y Pieter Spronck. *Monte-Carlo Tree Search: A New Framework for Game AI*. En *AIIDE*. 2008.

[ENC08] *Encog Machine Learning Framework*. https://github.com/encog, http://code.google.com/p/encog-java/, 2008.

[Epp] David Eppstein. *Computational complexity of games and puzzles*. School of Information & Computer Science, UC Irvine.

[EUG57] *European Go Federation*. http://www.eurogofed.org/, 1957.

[Fis96] George S. Fishman. *Monte Carlo: concepts, algorithms, and applications*. Springer series in operations research. Springer-Verlag, 1996.

[Gla04] P. Glasserman. *Monte Carlo methods in financial engineering*. Applications of mathematics. Springer, 2004.

[GS11] Sylvain Gelly y David Silver. *Monte-Carlo tree search and rapid action value estimation in computer Go*. *Artif. Intell.*, 175:1856–1875, July 2011.

[HA10] José Miguel Horcas Aguilera. *Apuntes de Inteligencia Artificial e Ingeniería del Conocimiento, Dpto. Lenguajes y Ciencias de la Computación (U. de Málaga)*, 2010. Capítulos 11, P11, 12, 13 y K.

[IGo82] *International Go Federation*. https://intergofed.org/index.php, March 1982.

[JAV] *Java para desarrolladores.* http://www.oracle.com/technetwork/java/index.html.

[MA08] Lawrence Mandow Andaluz. *Apuntes de Inteligencia Artificial e Ingeniería del Conocimiento, Dpto. Lenguajes y Ciencias de la Computación (U. de Málaga)*, 2008. Capítulos 8, 11, 12 y 13.

[Riv92] Ronald L. Rivest. *The md5 message-digest algorithm*. Internet RFC 1321, April 1992.

[SlB10] J. Sánchez, H. Álvarez, y D. Borro. *Gpu optimizer: A 3d reconstruction on the gpu using monte carlo simulations*. En *International Conference on Computer Vision Theory and Applications (VISAPP'2010)*, páginas 443–446. Angers, France, 2010.

[TT96] John Tromp y Bill Taylor. *Tromp-Taylor rules of go*. http://homepages.cwi.nl/ tromp/go.html, September 1996.

[Wei] Eric W. Weisstein. *Grundy's Game*. MathWorld–A Wolfram Web Resource.

Bibliografía

[All94] L.V. Allis. *Searching for solutions in games and artificial intelligence/ Louis Victor Allis*. Ponsen & Looijen, 1994.

[HD09] R.A. Hearn y E.D. Demaine. *Games, puzzles, and computation*. Ak Peters Series. A K Peters, Ltd., 2009.

[Las60] E. Lasker. *Go and go-moku: the oriental board games*. Dover Publications, 1960.

[MM08] José T. Palma Méndez y Roque Marín Morales. *Inteligencia Artificial: Técnicas, métodos y aplicaciones*. McGraw-Hill, 2008.

[Nil01] Nils J. Nilsson. *Inteligencia Artificial. Una Nueva Síntesis*. McGraw-Hill, 2001.

[Pap94] C.H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.

[Pre95] R.S. Pressman. *Ingeniería del software: un enfoque práctico*. McGraw-Hill, 1995.

[RN04] Stuart Russell y Peter Norvig. *Inteligencia Artificial: Un Enfoque Moderno*. Pearson Educación, 2 edición, 2004.

[SM04] S. Stelting y O. Maassen. *Patrones de diseño aplicados a Java*. PEARSON ; PRENTICE HALL, 2004.