Radikal Chess - Análisis y Diseño

Trabajo de Curso de la Asignatura de Fundamentos de los Sistemas Inteligentes
Grado II - EII - ULPGC
Curso 2013-2014
Fecha: 04/06/2014

Calvin Bulla
calvin.bulla101@alu.ulpgc.es

Dariel González Rodríguez
dariel.gonzalez101@alu.ulpgc.es

Resúmen

A continuación se expondrá el análisis, el diseño y la implementación del trabajo de curso para la asignatura Fundamentos de los Sistemas Inteligentes, que consta en la implementación de un juego denominado Radikal Chess, que se basa en las reglas del Ajedrez convencional.

1. Introducción al Problema

El juego se base en el ajedrez, modificando tanto el tablero, como las reglas del juego. De esta manera es necesario reevaluar representaciones de ajedrez ya establecidas.

Figura 1. El tablero modificado

Las reglas quedan modificadas de la siguiente manera:

- El primer movimiento de un peón siempre será de una casilla.
- No hay enroque.
- En los movimientos de no-captura, y salvo que sea para amenazar con un jaque, la pieza movida tiene que acercarse al rey rival. Esto no se aplica a los peones.
- La partida finaliza cuando uno de los jugadores pierde su rey o se queda sin movimientos válidos.

En este trabajo solamente vamos a plantear maneras de como resolver el problema representado. La implementación quedará pendiente para un trabajo consecutivo. Durante el texto a continuación además vamos a hacer referencia a conceptos y definiciones

expuestos en las clases de Fundamentos de los Sistemas Inteligentes.

Primeramente realizamos el análisis y modelado del problema, en cual exploramos diferentes maneras de representar los estados y operadores. A continuación haremos un estudio aproximado de la complejidad que ofrece el problema. Seguidamente nos dedicaremos a contrastar el diseño de la solución que consiste en encontrar los métodos de búsqueda más apropiados. Procedemos con una discusión del catálogo de heurísticas. La segunda parte del trabajo consiste en un informe de la implementación del juego, como descripción de su uso, la comprobación de su funcionamiento correcto y una presentación de experimentos realizados.

2. Análisis y Modelado del Problema

En esta parte vamos a elaborar como representamos los Estados y la información que deben contener, y los Operadores que generan nuevos estados para cada turno. Los puntos principales a tener en cuenta son facilidad de implementación, tamaño, y accesibilidad.

2.1 Análisis

Por un lado solo disponemos de un tiempo de desarrollo limitado, por lo tanto tenemos que ser capaces de generar resultados de prueba en un tiempo razonable. Por otro lado, la implementación de esta parte es crucial, ya que se encuentra en el camino crítico. Se ha demostrado, que intentar enseñar a la máquina jugar como un humano no es una tarea sencilla, y que en muchos casos hasta menos eficiente que la variante de fuerza bruta, que consiste en la generación de todos los estados successores posibles y la exploración de los mismos.

Esta exploración tiene dos factores que la limitan: El tamaño de un estado, mientras menos espacio consume, más podemos almacenar, y el tiempo de evaluación, mientras menos tardamos en explorar un estado, más podemos explorar en un tiempo razonable.

Tenemos que encontrar una manera de balancear estos factores para aumentar la frontera de exploración, ya que, hasta una heurística medianamente buena se puede convertir en una muy buena simplemente viendo más hacía el futuro. Nuestro objetivo es llegar a 4 ply por lo menos.

2.2 Modelado del estado

Con estas características del problema en mente procedemos a modelar el estado. Se trata de un juego con tablero, así que hay dos enfoques canónicos de como representarlo, una representación que se centra en las celdas y otra que se centra en las fichas en el tablero, cada una con sus ventajas y desventajas, por lo que hasta puede llegar a ser conveniente usar un modelo híbrido (aunque esto nos cuesta más espacio).

2.2.1 Representación por Celdas

En esta representación almacenamos información acerca de lo que se encuentra en cada celda del tablero. Esto se puede hacer mientras una estructura

matricial por ejemplo. Tiene la ventaja de que averiguamos el contenido de una celda O(1), pero la operación de evaluar todas las fichas de un cierto tipo en el tablero será de O(N).

2.2.2 Representación por Fichas

En esta variante en vez de las celdas almacenamos una lista para cada ficha, tanto de color blanco que de color negro, que contiene las posiciones de este tipo de ficha en el tablero. Para el tablero inicial esto sería por ejemplo:

Ficha	Posiciones
Peón blanco	A2, B2, C2, D2
Alfíl blanco	B1
Torre blanca	A1
Reina blanca	C1
Rey blanco	D1
Peón negro	A5, B5, C5, D5
Alfíl negro	C6
Torre negra	D6
Reina negra	B6
Rey negro	A6

Tabla 1. Representación por fichas inicial

Esta representación tiene el inconveniente de tener que recorrer todas las fichas para saber que objeto hay en una celda concreta (O(N) donde N es el número de fichas). No obstante resulta ser una representación que en extensión se suele usar mucho en los juegos de tablero, los así llamados Bitboards.

2.2.3 Bitboards

Los bitboards son una versión de la representación por ficha. En el ajedrez tradicional se basa en que hay un total de 64 celdas, por lo que la lista de posiciones cabe en una palabra de 64 bits (un long en Java), considerando que un bit activo (= 1) significa que la ficha se encuentra en dicha posición, y un bit no activo (= 0) el caso contrario. Como nosotros solo contamos con 24 celdas (4 x 6) hasta nos cabe en un una palabra de 32 bits (un int en Java). Para enumerar las posiciones podemos relacionar el bit menos significativo con la posición A1, y el bit 24 con la posición D6. Tenemos 8 bits no usados sin un valor concreto asignado.

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

Figura 2. Asignación de bits

De esta manera podemos conseguir que un tablero completo nos cabe en un array de 40 Bytes (= 10 Fichas x 4 Bytes cada una), una representación muy compacta. Tiene la ventaja que muchas queries que se le hacen al tablero se pueden codificar en bitboards también, y de esta manera se pueden realizar a través de operaciones lógicas y en O(1). Tiene la gran inconveniente que es más difícil de implementar, y que algunas queries son dificiles de hallar.

Existen varias maneras de extender la utilidad de los bitboards, como los bitboards rotatorios, que aparte del tablero canónico también almacenan versiones rotadas, para luego facilitar las queries de movimientos de torre y alfil, o los bitboards mágicos, que básicamente consisten en aplicar una función de hash para convertir el tablero en una representación más tratable para la query en cuestión.

2.2.4 Información adicional (-> véase 5.1)

Aparte de la representación del tablero es conveniente que el estado almacena información adicional. A continuación se exponen datos que pueden ser relevantes para evaluar el tablero por una heurística, o para facilitar la generación de movimientos.

Cabe destacar que algunos de estos puntos pueden ser redundantes o obtenidos a partir de la conversación de otro dato (como por ejemplo el turno y el jugador actual). También hay que tener en cuenta no almacenar demasiada información en el estado, ya que puede tener un efecto de polución (espacio vs. tiempo de ejecución).

Jugador actual o Turno
Está información es esencial para casi todos los aspectos del juego y por lo tanto estrictamente necesaria. Y se puede usar para modificar la heurística en función del tiempo.
Valor heurístico calculado
Puede resultar útil para la poda.
Número de peones
Para la heurística, aparte de eso se puede valorar su distancia a la fila contraria (= Promoción)
Todas las fichas de un color
Puede resultar útil para generar los movimientos. Una ficha no puede comer otra del mismo color
Posición del rey enemigo
Con este valor guardado se alivia la generación de movimientos, ya que no es necesario recorrer el tablero para averiguar su posición.
Valor en el tablero
Es sencillo de implementar y puede resultar muy útil para la heurística. Se puede actualizar cada vez que se quita o cambia una ficha.
Piezas atacando/atacados
Esto puedo resultar ser un dato complejo, ya que depende de los movimientos, por lo que habrá que evaluar el beneficio que tiene frente a su coste
Número de movimientos válidos
Véase el punto anterior. No obstante es importante para este juego, ya que una de las reglas es que se pierde sin movimientos válidos.

2.2.5 Make/Unmake

Uno de los límites en las búsquedas de fuerza bruta es la memoria. Entonces para no tener que realizar una copia entera del tablero planteamos un único tablero para toda la partida. Este tablero implementará una lista de movimientos realizados. De esta manera, para evaluar un movimiento, primeramente se realiza, se obtiene el tablero resultante, que se procede a evaluar, y finalmente se deshace el movimiento y el tablero vuelve a su estado inicial.

2.3 Modelado de los operadores

Si el estado en esencia es el tablero del juego, los operadores serán los movimientos de fichas. En principio hay estos cuatro tipos de movimientos

- Movimiento Normal
- Movimiento con Captura
- Promoción Normal
- Promoción con Captura

Cabe destacar que la promoción siempre será a reina, por el hecho de que no hay caballo como ficha disponible.

Aunque la representación de un movimiento es más bien fácil, su generación es una tarea complicada. Hay que tener en cuenta varios factores, los principales:

- Tipo de ficha
- Ocupación de las celdas
- Distancia al rey
- Jaque

Para acelerar la generación de movimientos podemos almacenar máscaras de generación, para así no volver a realizar cálculo redundante. Estas máscaras pueden contener movimiento de rey, alfil y torre, ya que el movimiento de reina se puede interpretar como un movimiento combinado de los dos anteriores.

3. Estudio de la complejidad

Como ya adresado en los puntos anteriores, dependiendo de como se plantea el problema puede llegar a ser elevadamente completo. Ya que nuestro enfoque es el de fuerza bruta, nos conviene más ser inexacto y vencer por masa, aprovechándose de la potencia de cálculo del ordenador. En este apartado se analiza la cantidad de estados que se van a generar en el árbol de exploración. Es difícil de prever, ya que depende mucho del camino que toma la partida. En general se define por

$$\#Estados = b^d$$

Donde b es el factor de ramificación y d es el nivel hasta cúal descendemos. En un juego ejemplar podemos llegar a una media de 6-7 movimientos por turno, con un total de 30 para acabar la partida, considerando que los dos jugadores juegan decente. Con estos datos se calculan un total de 2×10^{23} y 2×10^{25} estados explorables.

Esto solo son valores aproximados, que luego contrastamos con los valores de las simulaciones. También es importante saber, que muchos de los

movimientos son muy malos, así que la mayor complejidad del problema es encontrar una manera de podar de una forma eficiente y rápida. Aún así nos damos cuenta, que los estados que se puedan dar son muchos menos que los del ajedrez tradicional.

4. Diseño de la solución

Teniendo una visión general de cómo se modela el problema y que dificultades tiene, se puede proceder a diseñar algoritmos para elegir un movimiento para un jugador problema.

Se tratará de un algoritmo de exploración de árbol (los nodos serán los estados conectados por operadores). El algoritmo más simple para un problema de este tipo es el algoritmo minimax. Viendo la cantidad de nodos que se pueden generar es más favorable aplicar un método de poda. De esta manera podemos ampliar la frontera de exploración y jugar más “inteligente”.

4.1 Negamax

El negamax es una variación del Minimax. Se basa en evaluar un tablero solo para el jugador actual, y invertir el signo del valor evaluado al propagarlo a un nivel superior. En esencia realiza la misma búsqueda.

4.2 Búsqueda Alfa Beta

En una búsqueda con poda alfa beta, donde se guardan para cada nodo dos valores, alfa y beta, que servirán para obviar ramas que se sabe que no van a producir un resultado mejor. De esta manera ya se puede alcanzar una disminución notable de los nodos a explorar, como hemos visto en las prácticas anteriores.

La búsqueda alfa beta produce mejores resultados si los estados de entrada ya se encuentran ordenados. Eso depende mucho de cómo generamos los movimientos, pero aparte de eso también se puede implementar un algoritmo de bajada iterativa (Iterative Deepening) para dar una estimación previa de los nodos y ordenarlos consecuentemente. Habrá que evaluar en la práctica la eficiencia de este método, ya que en efecto evaluamos algunos estados repetidamente para podar otros. El factor de poda / reevaluación será el que decide si es razonable, junto con la ordenación de los movimientos.

4.3 Extensiones

Se pueden plantear diferentes tipos de extensiones que pueden facilitar la poda o la evaluación de un estado. Estos pueden ser parecidos a los que se usan en ajedrez.

4.3.1 Null Move

Si se supone que en una partida cada movimiento te resulta en una situación más favorable podemos usar el valor heurístico del estado actual para podar los hijos si producen una situación peor. En algunos casos eso puede llegar a ser una indicación mala, sobre todo cuando el jugador problema se encuentra

en jaque y no tiene otras opciones, pero en general puede generar resultados mejores y descartar muchos estados "tontos" de golpe. Aparte de eso es bastante sencillo de implementar.

4.3.2 Historia de movimientos

En algunos situaciones se da la casualidad de que un movimiento se repite varias veces (sin tener en cuenta la ficha). Por ejemplo primero mata alfil, luego mata reina. Es un factor auxiliar que entra en juego cuando un movimiento tiene una poda importante como consecuencia, porque genera un valor heurístico muy elevado.

4.3.3 Tabla de transposición

Muchas veces se puede dar el caso que hay más que una manera de llegar al mismo estado. Para no tener que reevaluar un estado repetidas veces se puede generar una tabla donde se almacenan los estados pasados y su respectivo valor heurístico. Previamente de evaluar un estado mediante la función heurística se puede comprobar si está en la tabla y usar este valor. Hay que anotar para que esto sea eficiente es necesario implementar un método para crear un valor de hash para un tablero para poder buscarlo en la tabla.

4.4 Anotaciones

Hay muchas maneras de afinar los métodos que se usan para generar un movimiento para el jugador problema y más que nada hay que evaluar cuáles son razonables de implementar. Eso también depende mucho de la representación que se ha elegido para el problema. Entonces para la primeramente nos vamos a centrar en implementar una búsqueda alfa beta y dejamos la extensiones planteadas para futuras versiones.

Aparte de eso es interesante introducir un factor de aleatoriedad en el juego, para que no se comporte de forma determinista. Eso se puede simular eligiendo una de las mejores soluciones en vez de la mejor siempre. Modificando la frontera de exploración podemos simular un nivel de dificultad para el jugador problema. Aparte de eso se pueden implementar diferentes heurísticas para variar su comportamiento.

5. Discusión sobre posibles funciones de utilidad

Si el método de búsqueda fuera el motor de un coche, la heurística sería la gasolina. De ella depende la capacidad que tiene el jugador problema para evaluar un estado. En este problema hay varios factores diferentes que influyen, por lo que trataremos de elaborar una heurística compuesta. Dependiendo del peso que le damos a cada factor (también en función al tiempo) variamos el comportamiento del jugador problema (agresivo, defensivo, ...)

5.1 Factores de evaluación(Heurísticas)

A continuación plantearemos algunas factores que ayudan para definir

la función de utilidad. Hay que tener en cuenta, que a la hora de implementarlas, algunos pueden resultar no viables, ya que no producen un resultado suficientemente valioso para el gasto que tienen a la hora de cálculo. Según la ley de amdahl el valor de una mejora parcial depende del impacto que tiene en el problema completo. Por lo tanto tratamos de ordenar los siguientes según la estimación del impacto que pueden tener el en juego. En la práctica no hemos implementado todas las heurísticas presentadas aquí y leves modificaciones de otras.

5.1.1 Valor en el tablero

Es bastante sencillo de implementar y puede dar una estimación muy buena de como esta la situación en el tablero. Con la ponderación de las piezas además podemos variar su funcionamiento.

5.1.2 Número de movimientos disponibles/viables

Está puede ser más difícil de implementar, sobre todo la segunda parte que además tiene que evaluar los movimientos según su viabilidad. Aún así se puede añadir como información al estado (en la búsqueda hay que hallarlos igualmente). También se puede discutir si no está ya indirectamente incluido en la búsqueda, pero en general y con las partidas jugadas se puede experimentar que el número de movimientos disponibles, hasta si son malos, sobre todo al principio, es un indicador bueno de quién está ganando el partido. Si un jugador no tiene más movimientos disponibles pierde.

5.1.3 Fichas en el centro

En el ajedrez tradicional ya es muy establecido que la mayor parte de la acción está en el centro del tablero, y que una buena posición ahí vale mucho. Se podrá evaluar si eso también se cumple para el Radikal Chess. Si nos pasamos en el modelado del estado con bitboards de hecho se puede implementar mediante un simple AND con una máscara.

5.1.4 Jaque

Está comprobación se tiene que hacer para cada estado igualmente, así que podemos aprovecharla como heurística auxiliar. Presionar el rey enemigo limita los movimientos disponibles del otro jugador y le condiciona en sus movimientos. No obstante muchos jaques, sobre todo al principio pueden resultar peligrosos para el jugador ya que se arriesga mucho a perder la pieza atacante.

5.1.5 Distancia al rey

Se puede comprobar que Radikal Chess es por si un juego muy agresivo, por lo que puede resultar conveniente atacar al rey a todo coste. Entonces la distancia al rey puede ser un factor útil. Un jugador agresivo puede darle mucha importancia. También hay que anotar que, ya que hay que acercarse al rey en la mayoría de los movimientos no agresivos, un jugador pasivo puede optar por no adelantarse mucho y dejar venir al otro. Entonces preferirá un valor alto

de distancia, ya que le deja más libertad a la hora de moverse y esperar.

5.1.6 Diferencia de fichas

Una vez que un jugador tiene una ventaja de fichas normalmente prefiere intercambiar fichas con igual valor, de está manera el otro jugador con cada jugada tiene menos opciones de mejorar su posición. Entonces puede ser un factor para variar la agresividad del jugador problema. Aún así está heurística se puede deducir de una combinación del número de movimientos y valor en el tablero.

5.1.7 Espacios abiertos

El tablero tiene un tamaño muy restringido. Entonces un espacio abierto favorece a un jugador que aún dispone de fichas claves como la reina, ya que puede fácilmente establecer un dominio sobre el tablero. Tiene mucho en común con la heurística anterior.

5.1.8 Peones cerca de la promoción

Esta heurística se puede implementar con un lookup en una tabla preparada, que contiene un valor predeterminado para una distribución de peones (se recuerda que las posiciones de los peones se pueden almacenar en un int, entonces se puede hacer uso de una tabla hash y ahorrar cálculo repetitivo). Aún así está heurística se obtener con una frontera de exploración suficientemente amplia y la heurística para evaluar el valor en el tablero (el cambio de peón por reina va a producir una mejora notable). Aparte de eso no es muy común que un peón puede llegar a la última fila.

5.1.9 Peones en el tablero

Se parece a la anterior, con la diferencia que es aún más sencilla y también tiene en cuenta que los peones se pueden mover independiente de donde está el rey enemigo, lo que permite una estimación de los movimientos posibles. Aún así el peón sigue siendo la ficha con el mínimo valor, así que está heurística puede falsificar el resultado.

5.1.10 Fichas atacando/atacados

Esta heurística en principio suena muy bien, pero tiene el inconveniente de que puede resultar muy costosa de calcular. Se parece bastante a cómo puede evaluar un jugador humano el tablero. Aún así se puede anotar que con las heurísticas anteriores ya se puede deducir indirectamente. Poder matar significa más movimientos disponibles y además cambios favorables en el valor de las fichas en el tablero.

6. Implementación

En el siguiente apartado procedemos con una descripción de la implementación del juego. Como ya hemos mencionado anteriormente se trata de un problema muy complejo, con muchas extensiones posibles. Aunque en los puntos anteriores hemos tratado de

presentar el problema de manera amplia, en la práctica nos tuvimos que centrar en los puntos más relevantes. Aún así abarcamos la esencia del juego, y de hecho el jugador problema es más que capaz de vencer a un jugador casual.

Aquí solo vamos a comentar el esquema general con cuál hemos estructurado el programa. El código fuente está comentado más en detalle. Empezamos con una vista general y luego la presentación de las clases en particular. La implementación se ha realizado en Java, y se debería poder compilar a partir de la versión 5.

6.1 Visión general

Hemos intentado estructurar el programa de forma modular para disponer de la posibilidad de extenderlo de diferentes maneras.

Primeramente tenemos la implementación de los objetos fundamentales, que incluyen el Color, la Pieza, la Posición, y el Movimiento. El tablero es la representación del estado inicial, y la heurística la función para evaluarlo. La generación de movimientos se realiza en una clase estática aparte. La búsqueda implementa el algoritmo Negamax con profundidad iterativa y poda alfa-beta y reordenación de movimientos. Es una búsqueda indeterminista, modificable en cuanto a profundidad máxima, tiempo máximo incertidumbre, y valores esperados. Un jugador representa a una entidad que, a partir de un tablero, realiza un movimiento. Aparte también se implementó una clase auxiliar para imprimir el tablero de forma legible. La tabla de transposición, y el generador de hash son clases auxiliares, que en la implementación actual no se usan, pero podrán resultar interesante en el caso de que se extiende el juego. Finalmente está la clase principal, que reúne todas las clases anteriores, establece una interacción por consola con el usuario, inicializa los jugadores problemas y el tablero.

6.2 Classes

A continuación se presentan en mayor detalle las clases relevantes, junto con algunas observaciones interesantes.

6.2.1 Color

El color de un jugador. Hemos optado por usar un enum, por que realmente se trata de una constante (Solo juegan blanco y negro). Sirve para identificar las piezas y para diferenciar los jugadores.

6.2.2 Pieza

Contiene la información necesaria para identificar una pieza en el tablero. Eso significa su tipo como su color. Cabe destacar, que no incluye información sobre su posición en el tablero. De esta manera se puede tratar más libremente ya que resulta innecesario actualizar este campo o crear nuevos objetos.

6.2.3 Posición

La posición identifica la combinación de fila / columna, como un índice, para situarla en el tablero.

6.2.4 Move

Es el desplazamiento de una ficha desde una posición inicial a una posición final con la opción de comer otra ficha. Esta información inicial se incluye, para facilitar los make/unmake en el tablero.

6.2.5 Board

El tablero contiene un mapa donde se almacenan las piezas en una posición. Su tamaño en principio es no variable. Está estructurado por celdas. Aparte de eso incluye información adicional, como el turno actual, o los valores de las piezas en el tablero. Adicionalmente también guarda la historia de movimientos. Se puede deshacer un movimiento previamente hecho. Después de cada cambio se tienen que actualizar los datos del tablero. Un tablero, junto a su historia de movimiento, se puede imprimir y leer a partir de texto, lo que nos permite guardar la partido seguir una guardada anteriormente. De hecho es así como se inicializa el tablero. Es importante anotar, que no se pretende tener más que dos tableros a la vez, siempre se trabaja sobre el mismo set de datos. Finalmente, el tablero genera un código de hash después de cada movimiento, para poder almacenar información acerca de si mismo en una tabla (aunque esto en la práctica se dejó planteada como una extensión posible).

6.2.6 Heuristic

Una heurística es una función que evalúa el estado para el jugador actual. Aparte de eso se puede generar una nueva instancia de la misma, asignándole un nuevo peso, y se pueden combinar varias heurísticas en una. En nuestra implementación está previsto, que la heurística devuelve un valor en el intervalo de [-20, 20] con valores extremos de 100, aunque su valor se puede extender.

6.2.7 Generator

En esta clase se realiza la generación de movimientos y los ataques (un ataque es un posición que queda amenazado por una ficha). Para mejorar el rendimiento, se precalculan los movimientos de torre y alfil y se almacenan en rayos. Así mismo se guardan las posiciones accesible para el rey. Está información se almacena en tablas que se pueden acceder a través de la posición de la ficha a mover.

6.2.8 AISearch

La búsqueda implementa “Iterative Deepening Negamax with Alpha-Beta Pruning”. Por ahora no hace uso de una tabla de transposición, pero si reordena los movimientos posibles después de cada iteración, para así mejorar la poda en la siguiente iteración. Por su estructura se limita por un tiempo y una profundidad máxima. Para afinar su comportamiento se puede especificar una ventana con los valores alpha y beta iniciales. Una vez que el algoritmo ha encontrado una

posición que considera como victoria, termina la búsqueda sin proceder a evaluar niveles más profundos. Basa su evaluación en una única heurística, pero esta a su vez puede ser una combinación de otras. En el nodo raíz de la búsqueda genera los hijos y se queda con el mejor, en los niveles inferiores solamente retorna la evaluación. Con los resultados de una búsqueda anterior reordena los movimientos, poniendo el mejor en primer lugar. Para obtener el indeterminismo, introdujimos un incertidumbre, que permite a la búsqueda elegir entre dos movimientos con evaluaciones parecidas. Permite la depuración a un fichero o por consola.

6.2.10 PrettyPrinter

Permite imprimir el tablero de forma legible. Además también ofrece las utilidades de resaltar piezas o movimientos.

6.2.11 ZobristHash

Genera números aleatorios, que se usan para la generación de Hash. Para más información sobre su funcionamiento se puede consultar el trabajo original.

6.2.12 TranspositionTable

La tabla de transposición por ahora solo guarda el valor y la profundidad de un estado evaluada. No se incluyó en los experimentos, ya que en la práctica resultó en resultados inesperados, posiblemente por colisión de hash. Aún así se deja planteada para incluirla en versiones futuras.

6.3 Main

Esta clase contiene la lógica principal del funcionamiento del programa y su interacción con el usuario. Se puede dividir en cuatro partes: Inicialización del tablero, de los comandos, y de la inteligencia artificial y finalmente el Read-Eval-Print-Loop.

6.3.1 Inicialización de la IA

Aquí se define el mapa de heurísticas en las cuales se pueden basar los diferentes algoritmos de búsqueda. Luego se pueden acceder a través de un nombre identificativo.

6.3.2 Inicialización del tablero

El tablero se carga a través del fichero “init.cfg” que contiene la configuración inicial. Aparte de eso se inicializan los diferentes jugadores, por un lado el jugador humano, pero también los jugadores problemas según su dificultad o comportamiento.

6.3.3 Inicialización de los comandos

En este apartado se implementan los comandos disponibles que se explicarán en la siguiente parte. Se acceden a través de un identificador con argumentos variables. Además existe un comando por defecto.

6.3.4 REPL

En este bucle se procesan los comandos introducidos por el usuario para luego imprimir los resultados obtenidos.

7. Manual de Uso

El juego se realiza por consola, no está implementado una interfaz gráfica. Los diferentes comandos están documentados en el propio código y también se puede acceder a través del comando help. A continuación describimos las diferentes utilidades que están implementadas:

Utilidades de tablero
Se permite imprimir el tablero, como averiguar las fichas en ciertas posiciones y movimientos disponibles para un color o una celda. Además se pueden des- y rehacer turnos.
Guardar/Cargar partida
Las partidas se pueden guardar en un fichero de texto o por la propia consola.
Ayuda
Se realiza una búsqueda para ayudar con la elección de un movimiento.
Lanzar partida
Se puede lanzar una nueva partida, especificando los dos contrincantes. A su vez se pueden lanzar una batería de juegos de una IA contra otra, almacenando sus trazas, para luego poder evaluarlas.

8. Estudio Experimental

A continuación se expondrán algunos de los experimentos realizados.

8.1 AI vs Human

En este experimento hemos jugado contra una inteligencia artificial con los parámetros de la tabla 8.1.1.

Profundidad máxima	5
Tiempo máximo	10s
Heurísticas	Valor de las piezas (peso=1), movimientos disponibles (peso=0.5)
Incertidumbre	0.5
Ventana	[-90,90]

8.1.1 Valores de la Búsqueda

Ya con está configuración es capaz de ganarnos, aunque un buen jugador puede vencerle fácilmente. Las medidas obtenidas se presentan en la tabla 8.1.2

Nodos expandidos	647.9
Nodos visitados	6451.9
Profundidad	5
Tiempo	136,7 ms
Podas Alpha Beta	468.7
Turnos	20

8.1.2 Resultados de la Búsqueda

8.2 AI vs Random

En este experimento hemos expuesto un jugador inteligente contra un jugador aleatorio con los parámetros de la tabla 8.2.1.

Profundidad máxima	11
Tiempo máximo	10s
Heurísticas	Valor de las piezas (peso=1), movimientos disponibles (peso=0.5)
Incertidumbre	0.5
Ventana	[-90,90]

8.2.1 Valores de la Búsqueda

Experimentamos, que a medida que avanza la partida, el jugador inteligente va ganando ventaja, hasta el punto que ya no baja hasta la profundidad máxima, ya que se da cuenta de que va a ganar. La tabla muestra las búsquedas que terminaron por timeout

Nodos expandidos	219658
Nodos visitados	3068179.2
Profundidad	8.7
Tiempo	10 s
Podas Alpha Beta	167678.5

8.2.2 Resultados de la Búsqueda

8.3 Contra otros equipos

Hemos lanzado nuestro jugador contra el jugador problema de otros equipos, usando las dos configuraciones. En ambos partidas ha salido vencedor, de hecho con un tiempo máximo de 10s es capaz de predecir una victoria hasta nueve turnos antes.

Experimentamos, que a medida que avanza la partida, el jugador inteligente va ganando ventaja, hasta el punto que ya no baja hasta la profundidad máxima, ya que se da cuenta de que va a ganar. La tabla muestra las búsquedas que terminaron por timeout

8.4 AI contra AI

Han competido dos búsquedas con diferentes heurísticas. La primera búsqueda equivale a la de 8.1.1, la segunda se puede ver en la tabla 8.4.1. Hemos incluido las trazas obtenidas en el código. En todos los casos ha ganado el jugador 1. (8.1.1)

Profundidad máxima	5
Tiempo máximo	10s
Heurísticas	#Piezas, #Ataques
Incertidumbre	0.5
Ventana	[-90,90]

8.4.1 Valores de la Búsqueda

9. Conclusiones

Como se trata de un problema de fuerza bruta, se puede optimizar mucho, basándose en la estructura que está por debajo. En nuestro código intentamos mantener pocos objetos en memoria, usando tipos básicos. Algunas mejoras, como la evaluación iterativa, o la reordenación de movimientos, al principio parecen introducir un coste adicional, aunque finalmente producen mejor resultado, como hemos visto en la competición contra otros equipos.

Un punto interesante acerca de las heurísticas es, que prevalecen las computaciones sencillas, ya que muchas veces nos permiten bajar hasta un nivel más, que luego resulta en una información mucho más detallada.

Quedan pendientes algunas mejoras, como son la implementación de una interfaz gráfica, afinación de las heurísticas, o la extensión mediante algoritmos más elaboradas para mejorar la búsqueda. Aún así podemos concluir, que ya con una esfuerzo computacional mediano, podemos vencer a un jugador humano.

10. Referencias

Transposition Tables, Bitboards, Alpha Beta Pruning - Gamedev.net, Francois-Dominic Laramee.
Una introducción a la programación de un engine de ajedrez.

Transparencias de clase
Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (3rd ed.).
El libro principal del curso, AIMA se basa en este libro.

Chessprogramming.wikispaces.com
Una página muy detallada con información acerca de diversos temas relacionados con la implementación programática de ajedrez. Fuente principal de muchas de las ideas y pseudocódigo.

11. Apéndice

Hemos incluido el los ficheros de fuente. Aparte de eso también hay una carpeta con algunas trazas realizadas. También se incluye un Ant - Buildfile.