ESTRATEGIAS DE PARALELIZACIÓN DE HEURÍSTICAS PARA EL JUEGO DE AJEDREZ

Eduardo Bouza Cubo, Daniel González Marcos, Jorge Pérez Barrio

FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

SISTEMAS INFORMÁTICOS

Curso 2013-2014

Directores:

José Ignacio Gómez Pérez
Christian Tenllado Van Der Reijden

Autorización de difusión

Los abajo firmantes, matriculado/a en Sistemas Informáticos de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Carrera: «ESTRATEGIAS DE PARALELIZACIÓN DE HEURÍSTICAS PARA EL JUEGO DE AJEDREZ», realizado durante el curso académico 2013-2014 bajo la dirección de José Ignacio Gómez y Christian Tenllado, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Madrid, 15 de agosto de 2014

Eduardo Bouza Cubo

Daniel González Marcos

Jorge Pérez Barrio

Agradecimientos

Eduardo:

-Agradecer a los directores del proyecto, José Ignacio Gómez y Christian Tenllado por su gran compromiso con el proyecto, guía y profesionalidad.

-A mi familia por su apoyo, ánimo y ayuda constante durante toda mi vida.

-Y a mis amigos y pareja que siempre están ahí para lo que haga falta y hacen que cada día sea un día único.

Daniel:

-A mis padres, que me han apoyado más allá de lo razonable este último año. Por todos los viajes de ida y vuelta en horas intempestivas en los que me han acercado al trabajo; por aguantar mi estrés y mal humor ocasional, y por todos las palabras de templanza sobre las relaciones personales. Especial mención a mi padre en lo académico y profesional, por ser un modelo de éxito a través del esfuerzo a pesar de la adversidad, de quien siempre me sentiré orgulloso.

-A nuestros directores, Nacho y Christian, por su inestimable conocimiento, esfuerzo, guía, y su interés en conocer minuciosamente las singularidades de la computación de ajedrez.

-Al muy insigne don Juan Miguel Lacruz Camblor, estudiante de Grado en Ingeniería Informática, por las llamadas y los ratos de dudoso esparcimiento prestado charlando sobre el proyecto, por su interés en el mismo, y por su incalculable aportación como ajedrecista.

-A Manuel Sánchez Pérez, estudiante de Grado en Ingeniería Informática, y más conocido por toda la facultad como 'Manu++','Señor++',o simplemente '++', por su valioso asesoramiento en las tentativas de implementación en C++. Sin duda, merece todos y cada uno de sus apelativos. Te deseo (y auguro) el mejor y más brillante de los futuros.

-A mis amigos, por la paciencia mostrada todo este año cuando les aburría con los pormenores del proyecto.

-A Jorge Pérez Barrio, 'Coque', amigo con quien he podido sobrellevar la incertidumbre, la frustración y los malos momentos este año, así como todos los anteriores.

Jorge:

-A mi Madre, en primer lugar, por ser mi baluarte. Porque detrás de todo lo que soy y he hecho está ella. Por 25 años demostrando sin descanso que Madre como la mía no existe otra igual. Por una devoción que ni yo muchas veces comprendo, por esto y mucho más, te quiero Madre.

-A mi Padre, porque cada consejo y recomendación suya ha sido un acierto existoso en mi vida. A un padre en el que me veo reflejado aunque no tanto como desearía. Por notar como pone todo su empeño y corazón en todo lo que a mí se refiere y por otras muchas cosas, te quiero Padre.

-A mi hermano, donde reside una figura ejemplar para mi desde mis comienzos a la vez que mi amigo más cercano. Agradecerte que a pesar de que pasan los años me siento como cuando éramos niños, eso para mí te hace indispensable.

-A 'la Abuela', en honor a una personalidad y actitud deslumbrante ante la vida. Por esa vitalidad y energía que hace que el término abuela me resulte insuficiente. Porque desde que tengo memoria he tenido la suerte de que estuvieras conmigo, abuela, tu nieto te quiere.

-Al Primi, en honor y agradecimiento a la más extraordinaria persona que he conocido. Por ser abuelo, amigo, arquitecto y constructor de mi maravillosa familia. Tu amiguete no te olvida.

-A Belén, el fichaje galáctico que completa mi familia. Porque a lo bueno uno se acostumbra rápido tengo la sensación de que eres de mi familia desde siempre.

-Al tío Carlos, por regalarme todos los años un puñado de buenos momentos.

-A la tía Carmen y la tía Saro, por todos los años de ayuda y apoyo.

-A mi gran amigo 'Karras', porque todo siga igual que hasta ahora.

-A mis amigos del barrio, miembros del Coffee-Pitt, amigos de fdi, al Águilas de Moratalaz, amigos del erasmus y compañeros de TPVs de Banesto, por hacer que una vista atrás de estos últimos años sólo me traiga buenos momentos.

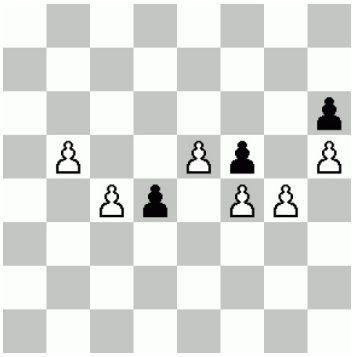
-A nuestros directores de proyecto, subrayando cada una de las palabras de agradecimiento de mis compañeros y recalcando una gran labor y una interminable paciencia.

-A Daniel González Marcos, amigo y compañero desde los comienzos en la facultad hasta el mismísimo último día.

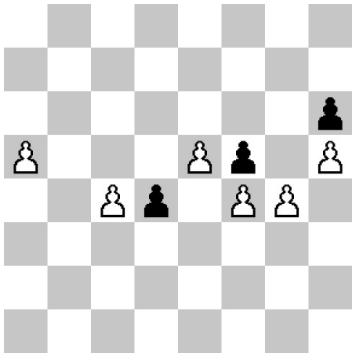
Índice general

Glosario de términios

Si bien el presente texto no pretende ser un tratado de ajedrez, tema sobre el cual hay abundante y dispar bibliografía, consideramos necesario glosar ciertos términos de ámbito meramente ajedrecístico, pues serán necesarios para la comprensión de las heurísticas y estrategias.



(a) Peón pasado



(b) Peón aislado

(c) Columna abierta

(d) Columna semi-abierta

Figura 1: Situaciones de juego relevantes

Peón pasado

Se denota por 'peón pasado' (ver Figura 1a) a aquellos peones que, en su dirección de avance, no tiene ningún peón enemigo ni en su columna ni en las columnas colindantes. En la imagen, el bando blanco tiene tres peones pasados, en b5,c4 y e5; mientras que el negro tiene un peón pasado en d4, y un peón pasado potencial si juega fxg4.

Peón aislado

Se denomina peón aislado (ver Figura 1b) a aquél que no tiene ningún peón de su propio color en sus columnas adyacentes. Los peones blancos de las columnas a y c están aislados, mientras que todos los peones negros lo están a su vez.

Columnas abiertas

Una columna abierta (ver Figura 1c) es una columna que carece de peones de ambos bandos. Tiene un importante valor estratégico, pues, controlada por una torre, multiplica las probabilidades de conseguir un ataque sobre la séptima fila, un elemento estratégico que puede determinar el curso de la partida. En el ejemplo, la columna e es una columna abierta, fácilmente controlable por el bando blanco.

Columnas semi-abiertas

Se conoce por columnas semi-abiertas (ver Figura 1d) aquellas que no presentan peones de un determinado color. En la ilustración, la columna e es una columna semiabierta para el blanco, y la columna d lo es para el negro. Pueden ser explotadas mediante el apoyo de una dama o torre. Además, son susceptibles (mediante ataque directo o cambio en casillas colindantes) de convertirse en una columna abierta.

Desarrollo (pieza)

Se entiende por el desarrollo de una pieza el primer movimiento efectuado por la misma.

Desarrollo (fase de juego)

El desarrollo en ajedrez, fuera del contexto del 'desarrollo de una pieza', se refiere a la fase del juego en el que siguen unos principios estratégicos básicos, generalmente alejados de un plan preestablecido. Estos dos conceptos fundamentales son el control del centro mediante peones y, fundamentalmente, el desarrollo de todas las piezas ligeras (caballos y alfiles), generalmente ubicadas atacando al centro o a piezas que ataquen o defiendan el centro, e intentando no mover ninguna pieza ligera más de un turno en las fases iniciales. El desarrollo prematuro de la dama supone un gran detrimento para el desarrollo. Finalmente, otro principio aconsejable es el de enrocarse antes del decimoprimer turno.

Medio juego (fase de juego)

Es la segunda fase del juego, que acontece tras la apertura o desarrollo, y precede al final. En esta segunda y más interesante sección, se intenta dar forma a un plan, que generalmente pasa por el ataque al centro. La ventaja posicional (mejor ubicación de las piezas) toma un carácter más relevante.

Final (fase de juego)

Última fase del juego. Se suele considerar como 'final' una partida tras 30 movimientos. Suele estar caracterizado por quedar pocas piezas en el tablero. Las damas, torres y alfiles cobran un mayor valor en esta fase del juego, pues, generalmente, tienen una mucho mayor movilidad. Los peones, especialmente si son pasados, adquieren todo su potencial, decantando las partidas a un lado o al otro. En los finales de peones, los reyes pasan de ser una cierta 'debilidad' que proteger a un elemento claramente ofensivo. Existen finales clásicos, como rey-torre contra rey; rey-dama contra rey; rey y dos alfiles contra rey; rey,alfil y caballo contra rey; y rey y dos torres contra rey, que han sido ampliamente estudiados, y cuyos algoritmos son conocidos. Para otros finales menos fácilmente catalogables, como los finales de peones en su conjunto, también se conocen principios teóricos cuya aplicación puede otorgar la victoria.

Apertura

En ajedrez, se denomina apertura a una secuencia de movimientos iniciales cuyas líneas principales son conocidas y estudiadas por los ajedrecistas. Para ciertas aperturas, se conocen todas las líneas aceptadas como buenas incluso a una profundidad de 25.

La Encyclopedia of Chess Openings (ECO) mantiene un estándar de catalogación.

FEN

El sistema de notación Forsyth-Edwards describe una determinada posición (esto es, una configuración de tablero).

Se describe como una cadena de texto, que consta de varios tokens separados por espacios en blanco:

- La matriz del tablero, fila a fila, desde el punto de vista del bando blanco.

- El turno del bando a quien le toca jugar (en inglés, 'w' o 'b').
- Posibilidad de enroque.
  - '-' representa que ningún bando se puede enrocar.
  - 'k' y 'q' representan, respectivamente, que las blancas pueden hacer enroque corto y largo.
  - 'K' y 'Q' representan, respectivamente, que las blancas pueden hacer enroque corto y largo.
  - Entre los caracteres k,q,K y Q puede existir cualquier combinación, según lo demande la posición.
- Puntero de captura al paso.
  - Si hubiera algún peón que pudiera ser capturado al paso, se denota la casilla inmediatamente detrás de él en notación algebraica.
  - '-' representa que no existe posibilidad de captura al paso.
- "Halfmove"
  - Representa el número de movimientos por bando desde la última vez que un peón fue avanzado o una pieza fue capturada.
- "Fullmove"
  - Indica el número de turnos jugados (entendiendo un turno por un movimiento de las blancas y un movimiento de las negras.

Por ejemplo, la configuración inicial del tablero sería representada por el siguiente FEN: rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

PGN

El PGN, Portable Game Notation, es un estándar de representación de partidas de ajedrez, orientado al uso por software y por humanos indistintamente. Se almacena en ficheros de extensión .pgn, que conforman un lenguaje de marcado. El fichero tiene dos secciones principales:

- Etiquetas de información de la partida.

    - Existen multitud de etiquetas, que se representan en el formato [nombre "valor"]
- Movimientos de la partida
    - Se representan en notación algebraica.

Sea un ejemplo:

[Event «Informal Game»]

[Site «London, England ENG»]

[Date «1851.07.??»]

[Round «-»]

[White «Anderssen, Adolf»]

[Black «Kieseritzky, Lionel»]

[Result «1-0»]

1.e4 e5 2.f4 exf4 3.Bc4 Qh4+ 4.Kf1 b5 5.Bxb5 Nf6 6.Nf3 Qh6 7.d3 Nh5 8.Nh4 Qg5 9.Nf5 c6 10.g4 Nf6 11.Rg1 cxb5 12.h4 Qg6 13.h5 Qg5 14.Qf3 Ng8 15.Bxf4 Qf6 16.Nc3 Bc5 17.Nd5 Qxb2 18.Bd6 Bxg1 19.e5 Qxa1+ 20.Ke2 Na6 21.Nxg7+ Kd8 22.Qf6+ Nxf6 23.Be7++ 1-0

Notación algebraica

Es un estándar de representación de secuencias de movimientos en ajedrez:

- Para representar el movimiento de un peón, se utiliza su columna y la fila que alcanza tras el movimiento.
    - a3 -Avance de un peón hacia a3.
    - axb3 -Notación para la captura de la pieza en b3 por parte del peón en la columna a.
    - axb6 -Tras 1.a4 h6, 2.a5 b5, 3.axb6 representa la captura al paso del peón negro en b5.
    - cxb8=Q -El peón blanco de la columna c captura la pieza en b8 y corona a una dama.

- En el caso de los enroques, se representan de la siguiente manera:
  - 0-0-0 -Enroque largo
  - 0-0 -Enroque corto
- Para el resto de piezas, se indica su tipo (K=rey,D=dama,R=torre,B=bishop,N=caballo) y su casilla de destino
  - Dh5 -La dama se desplaza a la casilla h5.
- En el caso de que el movimiento pudiera referirse a dos piezas del mismo bando, se desambigua de la siguiente manera:
  - -Si pueden distinguirse por su columna, se añade su columna (letra) tras el tipo de pieza.
  - -Si así no fuera, se distingue por la fila (número), insertándola tras el tipo de pieza.
- Para todo movimiento, si se da jaque al adversario, se añade el sufijo '+', y '++' para denotar el jaque mate.
- Finalmente, existen sufijos para denotar jugadas buenas, malas, sorprendentes, novedad teórica etc. Tienen un carácter relativamente subjetivo.

Notación descriptiva

Estándar de representación de secuencias de movimientos en ajedrez, hoy en desuso, en parte por la incapacidad que tiene de representar secuencias de movimientos en variantes de ajedrez con una configuración de tablero distinto al original.

Distancia de Manhattan

En contraposición a la distancia euclídea entre dos puntos, se define la distancia de manhattan como la suma de las diferencias absolutas de las coordenadas de ambos puntos. Es ampliamente utilizada en las ciencias de computación para tomar cotas aproximadas, pues calcular el camino mínimo entre dos puntos puede ser computacionalmente costoso, dependiendo del problema.

Capítulo 1

Introducción

El ajedrez, *Rey de juegos y juego de reyes*, ha fascinado a la humanidad desde su temprana aparición. Y no es en balde las pasiones que ha levantado a lo largo de los siglos, pues el trasfondo matemático subyacente al juego es prodigioso. Numerosos han sido los hombres de ciencias que se han visto en las garras del Juego, considerado por muchos conocedores como sencillamente «diabólico».

Adelantando el siguiente capítulo, cada avance tecnológico en la era de la información es empleado en el Juego, tratando de conseguir inteligencias artificiales más y más potentes, hasta el punto en que es esperable que en apenas 10 años no exista hombre capaz de derrotar a una máquina creada por el hombre. Donde, históricamente, el ajedrez había fascinado a gran cantidad de matemáticos, en las últimas décadas se ha creado una gran comunidad online de aficionados al ajedrez y profesionales de las ciencias de la computación, contribuyendo con su conocimiento en pos de la máquina perfecta. Pero, ¿por qué ese afán? ¿cuál es el reto que mueve la motivación?

No es en vano considerado como diabólico por sus conocedores, y opinión similar habrá de desarrollar todo ingeniero que estudie la problemática: el Juego es, computacionalmente hablando, indecidible en su conjunto.

La mera exploración del espacio de estados acotado en profundidad implica un cuidado exquisito en todos los detalles de la implementación, en continua búsqueda de la eficiencia, pues la explosión combinatoria en el ajedrez es, sencillamente, masiva. Sea, por ejemplo, la cantidad de nodos hojas en sucesivos niveles de profundidad exploratoria, esto es, turnos de juego:20,400,8902,197281,4865609,119060324... Una pregunta clásica en los estudiosos de las aperturas ha sido siempre «¿Cuál es la jugada óptima para empezar con blancas?». Pues bien, las matemáticas niegan la existencia de una respuesta formal a esa pregunta.

Sin embargo, en virtud a la constante evolución tecnológica y arquitecnónica de los computadores, ya es lejano el día en que las máquinas consiguieron explorar más líneas y a mayor profundidad que los humanos, superándonos, a día de hoy, con creces. A pesar de ello, la mera exploración no dota a la máquina de capacidad de juego, pues no es sino un rudimento necesario. La necesidad de «inteligencia» frente al «músculo» (la capacidad de exploración) es inherente al Rey de Juegos.

Pero, de nuevo, el ajedrez impone un nuevo problema computacional: ¿qué jugada es mejor que otra?. En juegos de estrategia más sencillos y de explosión combinatoria reducida, como el Blokus, se puede calibrar la bondad de un movimiento simplemente midiéndolo con los finales de partida esperados, pero ha quedado patente la imposibilidad de esto en el Juego. De este modo, no queda sino conferir a la máquina del conocimiento humano. A través de los años, se han desarrollado inteligencias artificiales en las que han colaborado numerosos ingenieros y ajedrecistas de élite, y se ha conocido de cerca la problemática: a pesar de que el ajedrecista posee ideas abstractas nacidas de la experiencia, es sumamente complicado dotar a la máquina de este conocimiento. Estas ideas abstractas (muy próximas al concepto de heurística) son disjuntas, y aplicadas a un dominio reducido. Sin embargo, a la hora de jugar una partida completa, la cohesión, ponderación y resolución de incongruencias entre estas ideas abstractas son sumamente difíciles de inculcar a una inteligencia artificial, donde un humano las calibra de forma natural, en numerosas ocasiones por instinto.

Nuestra aportación a esta duradera guerra de intelectos ha sido, desde su concepción allá por marzo del 2013, humilde y clara:

La simple implementación de una inteligencia artificial, robusta, sencilla en lo posible, legible y escalable; así como un protocolo y una interfaz gráfica de juego remoto entre humano-humano o humano-IA con las mismas características. Continuando sobre la anterior infraestructura, planeamos ahondar en la explotación del paralelismo y del cómputo distribuido para potenciar tanto la capacidad exploratoria como el motor heurístico. Sin embargo, en ningún momento albergamos el deseo de crear un monstruo ajedrecístico, pues desde el principio sabíamos que careceríamos de recursos y, especialmente, de tiempo. Sea, pues, un breve resumen al trabajo denodado de un año.

Capítulo 2

Trabajos anteriores y estado del arte

Desde el comienzo de la automática, antes de la era digital, siempre hubo un gran interés por desarrollar una 'inteligencia artificial' (incluso antes de que el mero término fuera acuñado).

Sean algunos hitos históricos especialmente relevantes, por uno u otro motivo:

- En el año 1768 se popularizó 'El Turco', un supuesto autómata que no resultó sino ser una estafa. El primer autómata capaz de jugar al ajedrez fue, en 1912, construído por Leonardo Torres y Quevedo, consiguiendo jugar finales de rey y torre contra rey.
- En la década de 1950 se desarrollaron los primeros programas capaces de jugar una partida completa.
- En 1966 tiene lugar, mediante telégrafo, el primer match entre software de ajedrez.
- En 1967, el software MacHack, desarrollado en el MIT, fue el primero en derrotar a un humano en un torneo, y el primero en conseguir un título oficial de ajedrez. Mirando retrospectivamente, la primera derrota en un ambiente de torneo supuso un punto de inflexión.

Debido al constante y drástico aumento de la capacidad de computación en los procesadores, tanto en ámbitos mainframe como en ordenadores personales, el software de ajedrez ha ido, a su vez, visto ampliada su capacidad de exploración y análisis de árboles de juego. Asimismo, con la introducción del multiprocesamiento, se han adaptado los algoritmos para explotar al máximo la capacidad de procesamiento del hardware actual. Esta revolución ha permitido observar la evolución de los resultados, desde los años 70, entre humanos y máquinas, donde en las últimas décadas la balanza parece comenzar a inclinarse hacia el

silicio. Quede este hecho constatado mediante la mención de algunas partidas especialmente significativas. En ellas, se indica primero el resultado en notación de ajedrez, tal que una partida ganada supone un punto y una partida en tablas supone medio punto.

Bent Larsen - Deep Thought, 1989: Por primera vez, una inteligencia artificial derrotó, en condiciones de torneo, a un Gran Maestro, y fue también la primera vez que el humano con mayor ranking ELO del mundo fue vencido por un software.

Kasparov - Deep Blue, 1996, 1997: 4-2 y $2\frac{1}{2}$-$3\frac{1}{2}$,respectivamente. Ésta última fecha fue la primera vez en que un software derrotó al actual Campeón del Mundo en un match a seis partidas en condiciones de tiempo de torneo. Hubo cierta controversia, sobre aspectos que otorgaban mayor ventaja a la máquina, como por ejemplo:

- A Garry Kasparov le fueron negadas partidas recientes de Deep Blue, mientras que ésta tenía una amplia biblioteca de su oponente humano.
- Entre las partidas, y acorde a las reglas preestablecidas, IBM ejerció su derecho de modificar el código fuente.
- Kasparov mantiene que, a partir de la segunda partida, observó cierta «creatividad», que siempre achacó a la intervención humana. Esto nunca pudo ser demostrado, y fue negado categóricamente por IBM.

Kramnik - Deep Fritz, 2002: 4-4, siendo Kramnik el actual campeón, tras derrotar a Kasparov.

Michael Adams - Hydra, 2005: $\frac{1}{2}$-$5\frac{1}{2}$, en el mismo año en que Michael Adams alcanzó el 6º puesto compartido en el mundial.

Kramnik - Deep Fritz, 2005: 2-4. En este match hubo unas condiciones establecidas, que no otorgan mayor ventaja a la IA: Kramnik recibió una copia del software aproximadamente un mes antes, para poder experimentar con él. El único cambio en la IA desde entonces fue la ampliación de la base de datos de aperturas. Adicionalmente, las tablas de finales almacenadas en Deep Fritz fueron limitadas a aquéllos finales de 5 piezas o menos.

Observando el pasado «reciente», se adivina un cambio sustancial en la forma de diseñar inteligencias artificiales en ajedrez:

En contra de usar arquitecturas especializadas, como el desmantelado Deep Blue que derrotó a Kasparov, se tiende a ejecutar sobre arquitecturas más estándar.

Por ejemplo, Deep Blue fue un supercomputador de arquitectura vectorial especializada, conformada por 30 nodos RS/6000, con un microprocesador P2SC a 120MHz cada uno y 480 coprocesadores especialmente diseñado para el cálculo ajedrecístico. Sin embargo, cuando Kramnik consiguió empatar a Deep Fritz en 2002, éste corría en una máquina octa-core.

De igual modo, en el «II People vs Computers World Chess Team Match»,Bilbao, 2005, Ponomariov, previo campeón del mundo, venció, de manera relativamente ajustada, a Fritz, que corría sobre un portátil de nivel usuario adquirido en una tienda local.

Asimismo, parece observarse un detrimento de la profundidad de exploración en contra de motores de inteligencia artificial más sofiticados. Por ejemplo, el Pocket Fritz 4, ejecutado sobre un teléfono móvil HTC Touch HD, con procesador a 528MHz, obtuvo un ranking ELO similar al de Deep Blue. Sin embargo, donde el gigante azul calculaba 200 millones de posiciones por segundo, el Pocket Fritz 4 calcula algo menos de 20.000 tableros por segundo.

Capítulo 3

Arquitectura de comunicación cliente-servidor

Como ha sido previamente mencionado, el pilar fundamental del proyecto consiste en un protocolo de comunicación que permita a dos personas jugar una partida remota. Éste protocolo está basado, como el nombre del capítulo indica, en una arquitectura cliente-servidor. Como un primer acercamiento, el cliente es la interfaz gráfica que utilizan cada uno de los jugadores, y el servidor un medio de comunicación entre ambos. Ambos términos serán explicados en mayor detalle.

Para la comunicación entre dos jugadores remotos, hemos optado por diseñar un protocolo de comunicación centralizado, tal que toda comunicación entre dos clientes es almacenada y encaminada a través de un servidor central.

En cuanto a la tecnología empleada para la comunicación en sí, hemos diseñado un protocolo de comunicación ad-hoc de alto nivel sobre sockets TCP-IP. Esta elección ha sido fundamentada por dos motivos principales:

- El mero deseo de implementar un protocolo de este tipo, algo que ninguno de nosotros había hecho durante la carrera.
- La enorme flexibilidad y especialmente portabilidad que proporciona, en contra de sistemas de invocación de métodos remotos existentes como java rmi.

De este modo, el cliente se conecta contra un puerto TCP-IP a la escucha en el servidor, tras lo cual éste crea un nuevo thread para atender al cliente, e intercambia con él mensajes pertenecientes a un repertorio cerrado en forma de texto plano.

Una vez recibido el mensaje (en adelante, 'paquete'), el servidor lo procesa y almacena, en espera de comunicarle la respuesta al cliente que así lo solicite.

3.1. Arquitectura de comunicación

Todos los paquetes siguen el mismo formato: cadenas de texto separadas por tabuladores, correspondiendo el primer token al tipo de paquete [1]

Sea pues el repertorio de mensajes, así como su dirección:

client → login version user pass	→ server
client ← loginAnswer { OK/WRONG }	← server
client → logOut	→ server
client → queryClientPool	→ server
client ← queryClientPoolAnswer [user-nick {y/n} \ n]	← server
client → refreshMatches	→ server
client ← refreshMatchesAnswer [white black date [movement FEN]\ n]	← server
client → refreshMatchesLastMovement	→ server
client ← refreshMatchesLastMovementAnswer [white black date #mov movement FEN]	← server
client → refreshChallenges	→ server
client → challenge adversary-nick my-colour FEN	→ server
client ← challengedBy adversary-nick your-colour FEN	← server
client → challengedByAnswer {accepted/refused} challenger-nick	→ server
client ← challengeAnswer {accepted/refused} your-colour adversary-nick FEN	← server
client → sendMovement FEN algebraic-movement adversary-nick	→ server
client ← ERROR further-information	← server

Y una explicación detallada de los paquetes:

[1] En el diseño original, se planeó reservar un primer byte en formato numérico para el tipo de paquete, tal que las comprobaciones de strings en el server-side fueran sólo las imprescindibles. Sin embargo, se ha mantenido el formato de texto plano por comodidad a la hora de depurar.

login: Una vez conectado el cliente al socket del servidor, un cliente adulterado, o un software desarrollado mediante ingeniería inversa para interactuar con el protocolo, podría mandar mensajes. Sin embargo, todo mensaje será descartado hasta que el client-side se autentifique correctamente mediante este paquete.

Los tokens del paquete son la versión de compilación del cliente (curándonos en salud ante futuras modificaciones mayores), su usuario y su contraseña, que a día de hoy viaja en texto plano.

loginAnswer: El servidor contestará al client-side, informándole de si ha sido satisfactoriamente autentificado o no. Aunque no se ha contado con tiempo para implementarlo, sería sencillo añadir una capa de seguridad para evitar ataques por fuerza bruta.

logOut: Mediante este paquete, el cliente informa al servidor de su desconexión, permitiendo hacer ésta de manera ordenada.

queryClientPool: Con esta petición, el servidor le enviará un paquete queryClientPoolAnswer, refrescando los usuarios conectados al sistema.

queryClientPoolAnswer: El formato es el siguiente: queryClientPoolAnswer, y una lista de tuplas con el nombre del cliente; un carácter (y/n) que indicará si el cliente está actualmente o no; y un salto de línea como tercer y último campo de la tupla.

refreshMatches: Este paquete representa la petición, por parte del client-side, de refresco de toda la información concerniente a todas sus partidas abiertas, es decir, tanto la configuración actual del tablero, como todo el histórico de movimientos.

Supone un cierto peso en el procesamiento tanto para el servidor como para el cliente, con lo que sólo es llamado cuando es realmente imprescindible, esto es, justo después de la autentificación satisfactoria del cliente.

refreshMatchesAnswer: La respuesta por parte del servidor va expresada como una lista de partidas separadas por saltos de línea. Una partida está representada mediante una tupla, indicando el nick del jugador blanco, el nick del jugador negro, la fecha de creación de la partida, y una lista de movimientos. Los movimientos son, a su vez, tuplas conformadas por el movimiento expresado en notación algebraica y su FEN resultante, separados mediante tabuladores.

refreshMatchesLastMovement: Una vez que el cliente ha recuperado todo el historial de movimientos de sus partidas abiertas, es redundante y computacionalmente costoso

forzar tanto al server-side como al client-side a procesar una información innecesaria. De esta manera, el mensaje refreshMatchesLastMovement solicita el refresco del último movimiento de cada una de las partidas abiertas.

refreshMatchesLastMovementAnswer: Análogamente al paquete refreshMatchesAnswer, refreshMatchesLastMovementAnswer proporciona una lista de las partidas abiertas por el cliente, separadas por salto de línea. Las partidas, a su vez, contienen los campos nick de las blancas, nick de las negras, fecha de creación de la partida, el número del último movimiento registrado en formato halfmove, el movimiento en sí expresado en notación algebraica y su FEN resultante.

refreshChallenges: Mediante el envío de este paquete, el cliente solicita por respuesta tantos paquetes challengedBy como peticiones de partida tenga pendientes.

challenge: Este paquete representa la petición de un cliente de desafiar a otro (que puede ser un perfil de inteligencia artificial). En él, se indica el nick del adversario, el color con el que jugará quien desafía, y un FEN inicial que, si bien no es todavía configurable, podría escalarse el client-side para permitir jugar con handicap o en otras variantes de ajedrez.

challengedBy: El servidor notifica a un cliente una petición de desafío pendiente. Se indica quién le desafía, con qué color habría de jugar el desafiado, y el FEN inicial de la partida.

challengedByAnswer: Este paquete modela la respuesta de un cliente que ha recibido un desafío. Se indica si ha sido aceptado o declinado y el nick de quien le desafío, para desambiguar el caso en que un cliente tuviera varios desafíos de sendos adversarios distintos.

challengeAnswer: El servidor responde a un cliente que hizo una previa petición de desafío. Le informa de si ha sido aceptado o rechazado, el color con el que debería jugar, el nick del adversario y el FEN inicial.

sendMovement: Mediante este paquete, el cliente reporta al servidor un movimiento que acaba de efectuar en una de sus partidas abiertas. La comprobación semántica del mismo (esto es, comprobar la validez del movimiento desde el punto de vista de las reglas del ajedrez) es efectuada tanto en el client-side como en el server-side, para

evitar ataques de ingeniería inversa contra el protocolo. En el mensaje se indica el FEN resultante, el movimiento expresado en notación algebraica, y el nick del adversario.

ERROR: Este paquete, siempre enviado desde el servidor hacia los clientes, le reporta de algún error semántico en la construcción del paquete. Si bien nos ha sido de utilidad a la hora de depurar, en general sólo ocurriría en clientes cuyo código estuviera adulterado, o en otro software creado de cero para interactuar con el protocolo.

Como puede observarse a la luz del anterior protocolo, modela un comportamiento master-slave del cliente frente al servidor, tal que toda comunicación del servidor hacia el cliente es originada por una petición previa en sentido contrario.

De este modo, el cliente envía periódicamente peticiones de refresco sobre los últimos movimientos de cada una de sus partidas abiertas, de los clientes conectados y de las invitaciones a partida que pudiera tener pendientes.

Una de las ventajas de haber elegido la construcción de un protocolo basado en paso de mensajes sobre sockets TCP-IP es su escalabilidad: si bien no hemos implementado un paquete 'resign' o 'drawOffer', para rendirse y ofrecer tablas, respectivamente, su adición sería trivial sobre la infraestructura ya existente.

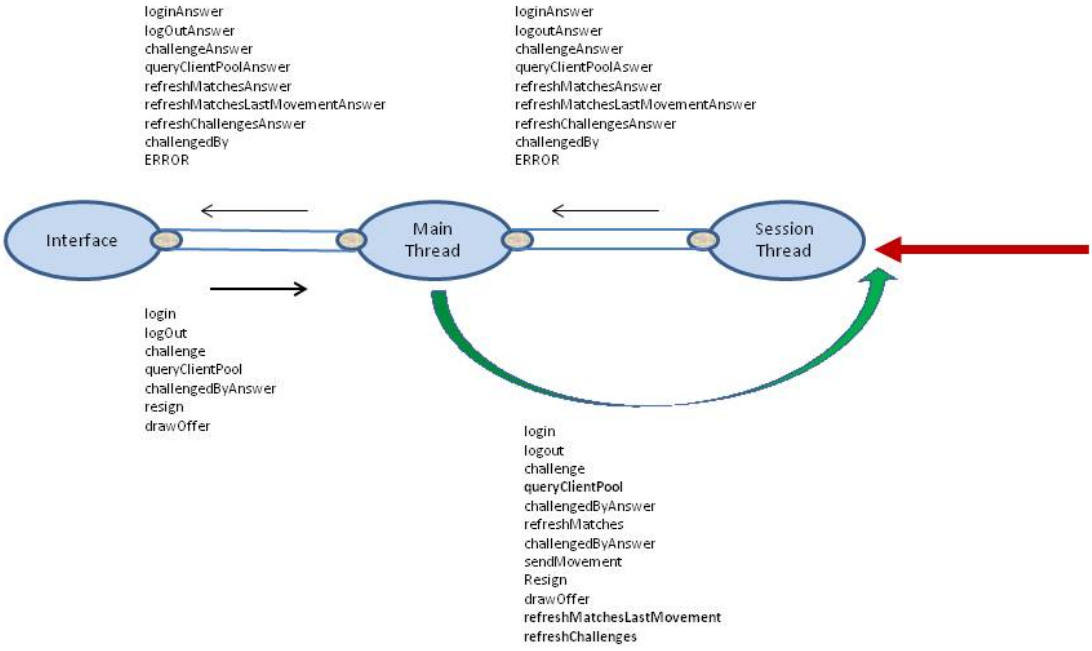


Figura 3.1: Arquitectura del cliente

En la figura 3.1 representa la arquitectura del client-side, representando cada óvalo un thread, los cilindros canales de comunicación internos entre hilos, la línea roja el socket contra el servidor, y la línea verde una escritura al socket por parte de un thread.

Los paquetes marcados en negrita son aquéllos que son periódicamente enviados por el client-side para refrescar información.

Como puede ser inferido a raíz de la imagen, la arquitectura del cliente sigue un Modelo Vista Controlador, que desacopla la interfaz humana de la lógica de comunicación, lo que haría relativamente sencillo construir una nueva interfaz.

3.2. Arquitectura del servidor

Como ya ha sido comentado, el servidor sigue un modelo multi-thread, tal que crea un hilo para atender a cada nuevo cliente. A la luz de la anterior descripción del repertorio de mensajes, se adivina que el servidor lleva a cabo una tarea de mera comprobación sintáctica y semántica de los paquetes, y una lectura o escritura contra su base de datos (salvo a la hora de generar un movimiento por parte de un perfil de inteligencia artificial, que será analizado en capítulos posteriores).

El hecho de haber diseñado una arquitectura de comunicación sobre paso de mensajes en texto plano mediantes sockets TCP-IP permite que el servidor pueda ser implementado en un lenguaje de programación distinto al del cliente. En concreto, y aunque ambos están en java, se planeó la migración de un subconjunto del motor a C++, como será detallado en el capítulo de conclusiones.

Así pues, definamos primero la base de datos, aspecto central del servidor. Ésta subyace en un sistema MySQL, local por eficiencia, pero que bien podría ser remoto.

3.2.1. Arquitectura de la base de datos

En la figura 3.2 se muestra el gráfico del modelo relacional diseñado.

El diseño ha sido preparado para poder hacer una explotación en aras de un futuro machine learning. Como puede observarse, los usuarios participan en las partidas con dos roles, como jugadores de blancas o negras. A su vez, las partidas contienen movimientos, y los resultados válidos pertecen a un catálogo determinado.

Si un usuario tiene su flag is-human a false, tendrá entradas en las tablas de machine-users-engine y machine-users-config, que indican, respectivamente, los motores de exploración de ese perfil (que se ejecutarán en orden secuencial hasta que uno devuelva un mo-

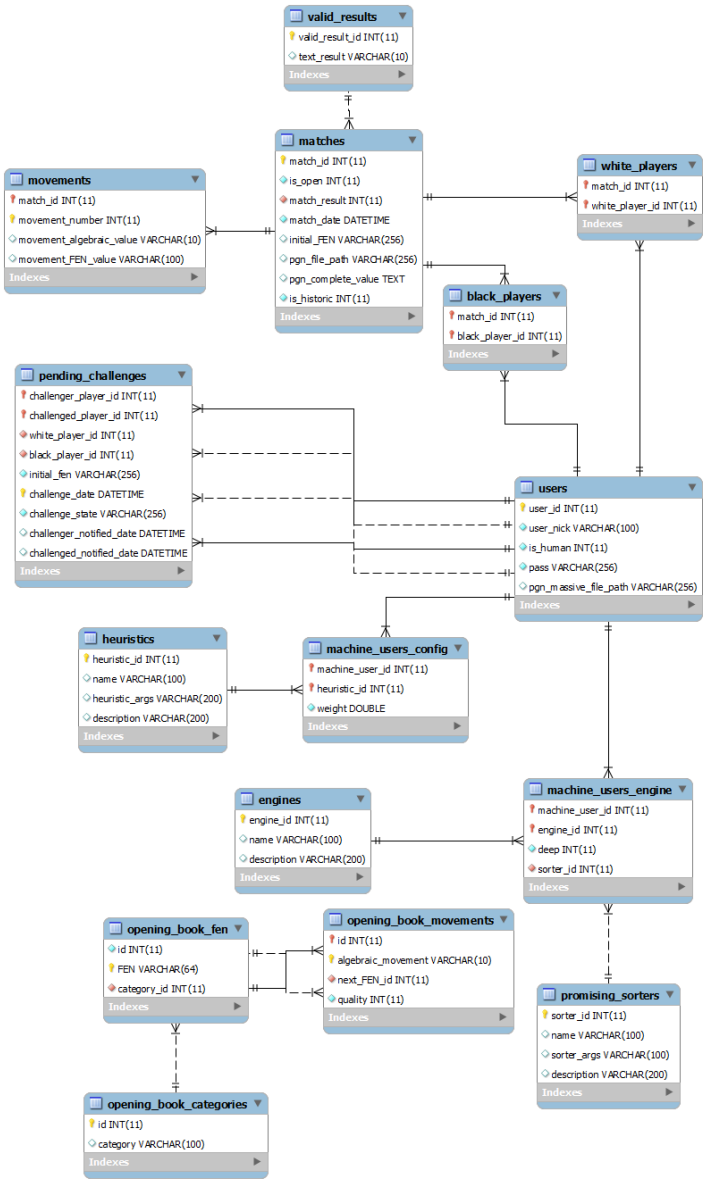


Figura 3.2: Modelo Entidad Relación de la Base de Datos

vimiento) [2] y el perfil heurístico que utiliza (las heurísticas que usa y sus constantes de ponderación asignadas).

Asimismo, cada uno de los motores tiene un promising-sorter asociado. Mediante las entradas en la tabla promising-sorters se indica la forma en la que los motores ordenan los hijos a nivel 1 de un nodo antes de pasar a expandirlos. A día de hoy, puede ser la no-ordenación en absoluto (respetar el orden de generación natural) o una ordenación aleatoria, que ha arrojado mejores resultados en el motor con poda alfa-beta.

Finalmente, existen 3 tablas, desacopladas del resto del modelo, que almacenan el catálogo de aperturas conocidas por el sistema, estructuradas de tal modo que sirvan tanto de sistema de estudio de aperturas por parte del usuario como para decidir el siguiente movimiento más prometedor por parte del motor de aperturas.

[2] Como se detallará en el capítulo 4 , existen motores de exploración que pueden no devolver ningún movimiento.

Capítulo 4

Motor de juego

En este capítulo se procede a dar una descripción sobre el motor de juego, concretamente nos centramos en el núcleo del motor. Como núcleo del motor de Juego nos referimos al conjunto de métodos responsables de garantizar la cohesión del juego y el correcto funcionamiento de una partida.

Comentar antes de nada que el núcleo del motor es común tanto para el cliente como para el servidor en cualquiera de sus variantes de juego, con la diferencia de que en la parte del servidor se invoca en el ámbito de acción de la inteligencia artificial mientras que en el cliente existe una interfaz gráfica.

4.1. Motor de juego

Durante el análisis previo a la implementación se decidió que eran tres los componentes elementales en una partida de ajedrez:

1. Tablero
2. Pieza
3. Casilla

Hemos basado el diseño del núcleo en este modelo, con lo que perseguimos una organización del código clara, organizada y modular, en la que sea fácil de localizar o colocar cualquier comportamiento que se desee buscar o incorporar en un futuro.

Son estas tres entidades las encargadas de todo el comportamiento y la distribución del núcleo del motor de juego. Estas entidadades completan su comportamiento cumpliendo en

muchos casos la función de adaptar o coordinar al núcleo del motor las distintas situaciones desde las que se invoca al motor.

Antes de comenzar con la descripción específica de cada elemento del núcleo, es importante definir una serie de términos que consideramos importantes llegado a este punto.

Ya que el motor se basa en la notación algebraica de la ajedrez, para poder garantizar que un movimiento algebraico es válido tiene que cumplir con todos y cada uno de los siguientes términos:

- Movimiento léxicamente válido: El movimiento generado o recibido en el motor de juego debe cumplir la gramática definida de la notación algebraica.
- Movimiento sintácticamente válido: El análisis sintáctico se cerciora de que toda pieza se mueva de manera correcta, teniendo en cuenta su naturaleza y la situación de las casiilas a las que implica cualquiera de sus movimientos definidos.
- Movimiento semánticamente válido: El análisis semántico de un movimiento implica que pese a que un movimiento puedes ser tanto léxico como sintácticamente válido no tenga un sentido válido en el contexto de la partida. Las situaciones como las que se ha ejemplificado ocurre cuando un movimiento correctamente notado, y perfectamente válido sintácticamente pone a su propio rey en jaque.

Concluir con que estos tres análisis son realizados a diferentes niveles del motor de juego. Las tres entidades del motor de juego son:

4.1.1. Keyboard

Entidad que representa el tablero de partida. El comportamiento fundamental de esta clase está implementada en los siguientes métodos que se describen a continuación.

parseALgebraicMovement: Es el método que regula si un movimiento es completamente válido (léxico, sintáctico y semántico). Es interesante destacar que es a este nivel donde se aplica el filtro sámantico para comprobar si un movimiento es válido o no, es decir, es aquí donde se comprueban y regulan las condiciones de jaque de la partida en juego. Se dispone de una lista de todos los movimientos sintácticamente válidos sobre los que aplica un análisis semántico. En caso de acierto se aplica el movimiento.

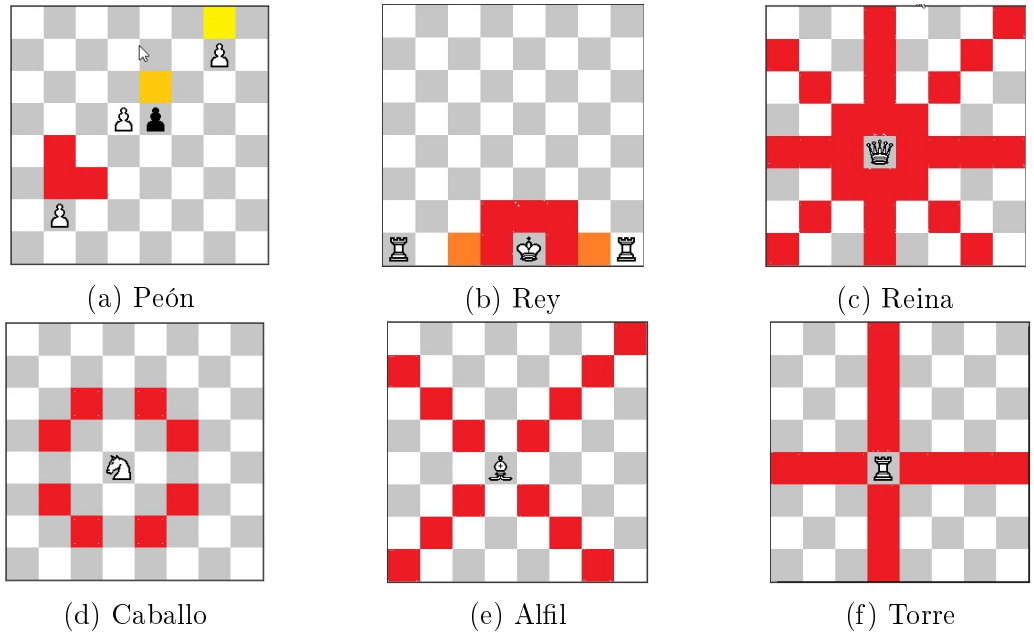
allPosibleMovements: Método que genera todos los movimientos semánticamente válidos que el conjunto de piezas restantes de un usuario es capaz de realizar. El contexto de acción es el Tablero, aunque depende únicamente de Pieza y Casilla.

4.1.2. Piece

Entidad correspondiente a una pieza del juego. Los métodos fundamentales de esta entidad en el núcleo del motor de juego son los expuestos a continuación.

validMovements: Genera todos los movimientos sintácticamente válidos para una pieza. El ámbito de este método es más reducido, ya que sólo depende de la naturaleza de la propia pieza y la situación de su casilla y posibles casillas afectadas.

generateAllDestinySquares: Es el método que garantiza el correcto significado sintáctico de los movimientos algebraicos. Recorre todas las posibles casillas destino para una pieza dependiendo de su naturaleza, descartando casillas en tanto en cuanto dicho movimiento no sea sintácticamente posible. La figura 4.1 muestra la movilidad posible para cada pieza (las situaciones de tablero son irreales, se pretende mostrar la movilidad de la pieza).



(a) Peón (b) Rey (c) Reina

(d) Caballo (e) Alfil (f) Torre

Figura 4.1: Ejemplos de posibles movimientos de piezas en tableros

algebraicMovementGenerator: Este método es el que regula la corrección léxica de todos los movimientos algebraicos que circulan por el motor de juego. Recopila información de un movimiento (pieza, casilla origen, casilla destino) y genera el movimiento algebraico gramaticalmente válido. La propia gramática de la notación algebraica obliga a que este método se divida en los tres siguientes:

- algebraicMovementConstructor: movimiento normal de pieza.
- algebraicCaptureConstructos: movimiento en captura de pieza.
- algebraicCaptureEnPasssantConstrutor: movimiento de captura al paso para peones.

4.1.3. Square

Esta entidad representa al elemento más atomico definido en el ajedrez, que son las casillas. Sin embargo cumple una importantísima función en el motor de juego, ya que para determinados movimientos «más peculiares» en una partida de ajedrez, como pueden ser la captura al paso de un peón, los enroques o las coronaciones, el motor las filtra y trata de manera específica gracias a las «casillas especiales».

Estas «casillas especiales» son determinadas en el tablero por su posición, es decir, existen determinadas casillas que podrán tomar un rol de casilla especial y otras muchas, de hecho la gran mayoría, que no lo harán nunca. Esto es debido a que las situaciones especiales del juego para las cuales utilizamos esta funcionalidad de Square están vinculadas al posicionamiento espefíco de ciertas piezas en el tablero. En la figura 4.1a se marcan las casillas especiales a las que nos referimos para el peon. En la figura 4.1b estan marcadas las casillas especiales para el rey.

De esta manera gestionamos las situaciones especiales del juego antes mencionadas, tratadas de esta manera peculiar dada su irregular aparición en la vida de una partida.

A continuación una vez finalizado con el núcleo del motor de juego, queremos dar una explicación del modo de uso que tiene desde los ambitos de acción del cliente y del servidor. Para ello consiramos apropiado dar a paso a la interfaz gráfica y a los generadores de movimientos.

4.2. Interfaz

Como punto de partida decidimos que, puesto que la componente gráfica no era un objetivo del proyecto, la interfaz debía de ser lo más simple e intuitiva posible.

Dicho esto, estructuralmente elegimos una distribución de tipo borderLayout, compuesta de una zona principal centrada rodeada de areas Norte, Sur, Este y Oeste que habilitan la posibilidad de añadir más componentes visuales. La figura **??** imagen muestra la distribución descrita.

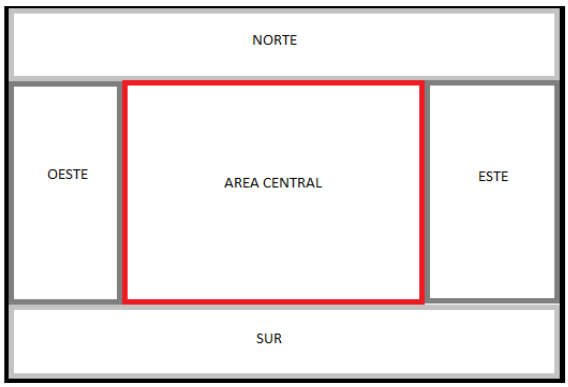


Figura 4.2: distribución borderlayout

En lo que se refiere a la funcionalidad de la interfaz se divide en dos módulos:

4.2.1. Window

Se corresponde con el frame principal de la aplicación, donde se gestiona la distribución gráfica de la aplicación.

En la parte superior del frame se ancla una barra de menús desplegables ocupada de lanzar acciones disponibles a nivel usuario. La estructura gráfica de áreas donde van los paneles contenidos en el frame es la siguiente:

- Área central: Situado en el centro del frame, este panel acapara la actividad de mayor importancia de la interfaz, de ahí su estratégica localización. Mediante un panel tabulado es capaz de gestionar la funcionalidad definida que permite a un usuario mantener varias partidas activas simultáneamente. En este panel tabulado es donde se sitúan entidades Interface.
- Área norte: Este panel establecido se encuentra vacío. Queda habilitado en la estructura de la interfaz para futuros usos o ampliaciones.
- Área Sur: Este panel inferior es utilizado para publicar mensajes a usuario.
- Área Este: El panel de la derecha a su vez está dividido en dos secciones: histórico de movimientos de la partida activa en el área central y listado de usuarios conectados a tiempo real.

- Área Oeste: Este panel establecido se encuentra vacío. Queda habilitado en la estructura de la interfaz para futuros usos o ampliaciones.

Actualmente la interfaz sólo está utilizando el área Sur y Este para añadir funcionalidades y mostrar información en el servicio de presentación. Se puede observar la estructura visual actual del Frame debido a la no utilización de todos sus componentes en la figura 4.3.

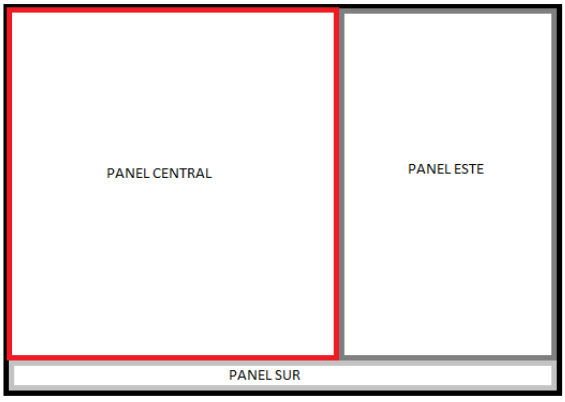


Figura 4.3: Distribución estructural de la interfaz

Añadir que en ciertas ocasionas los mensajes al usuario son lanzados a través de ventanas emergentes por claridad, ya que consideramos que son de más importancia que los normales mostrados por el panel inferior.

4.2.2. Interface

Clase en la que se delega todo el comportamiento funcional orientado al juego. Se trata de un canvas en el que se plasma el tablero de ajedrez con la situación de partida actual en juego, rotado según el color elegido por usuario.

Mediante captura de eventos de ratón es capaz de asimilar el movimiento especificado por el usuario. Tras realizar el análisis de interpretación de movimiento del cliente se invoca al nucleo del motor de juego. Concretamente una vez determinado origen y destino del movimiento se construye el movimiento algebraico con algebraicMovementGeneratos y luego se procede al análisis léxico, sintáctico y semántico con la llamada a parseAlgebraicMovement.

En el caso de no ser el turno de movimiento del cliente, el hilo MainThread transmite

el movimiento enviado por el servidor a la interfaz y se procede al análisis con parseAlgebraicMovement.

Si el movimiento resulta ser válido, es desde aqui desde donde se comienzan a desencadenar la consecución de eventos que se producen al ser realizado un movimiento de partida válido por parte del cliente, hablamos de eventos de comunicación mediante la arquitectura cliente-servidor.

4.3. Generadores de movimientos

En la parte del servidor el núcleo del motor de juego es utilizado para la generación de movimientos automáticos. Esto es necesario para poder jugar contra usuarios computados, es decir, contra la máquina.

Una primera aproximación a lo que en este módulo se realiza es la siguiente: mediante un algoritmo que envuelve a llamadas al núcleo del motor de juego se consiguen generar todos los movimientos posibles a partir de una situación de tablero desde ese turno hasta n turnos después.

Para entender un poco más a fondo la elaboración de movimientos por parte del servidor fragmentamos el proceso en tres puntos bien desacoplados:

Motor de juego Dado un tablero, genera todos los posibles movimientos a nivel 1. Está implementado en Keyboard::allPossibleMovements.

Motor heurístico Desacoplado de los motores de exploración de subárboles, es el encargado de la evaluación de los tableros explorados, y representa la toma de decisión de un camino respecto a otros. Este apunte sirve como preludio ya que será estudiado en el siguiente capítulo.

Motor de exploración de subárboles Genera todos los posibles movimientos hasta turno n de partida. Es la capa que despliega un subárbol a una cierta profundidad acotada. Actualmente existen cuatro implementaciones distintas:

- motor basado en algoritmo iterativo sin poda.
- motor basado en algoritmo recursivo con poda alfa-beta.
- motor de aperturas (leídas de la base de datos central).
- motor de *jugadores históricos* (leídas de la base de datos central).

Sobre estos dos últimos, funcionan de manera muy similar: cuando le toca mover a la IA, consulta en la base de datos (libro de aperturas o en la tabla partidas, dependiendo de si es el motor de aperturas o de «jugadores históricos», respectivamente), tratando de localizar una entrada que concuerde con los cuatro primeros tokens (esto es, la matriz de piezas, el turno de quien le toca jugar, la posibilidad de enroque y el puntero de captura al paso) del FEN del tablero sobre el cual tiene que decidir su movimiento.

En el caso del motor de aperturas, para todo tablero presente en el sistema, existen punteros a determinados hijos que conformen líneas conocidas (ya sea por ser éstas habituales, buenas, o francamente malas), ponderados someramente según la calidad del movimiento. De este modo, elegirá el hijo con mejor valor heurístico.

El motor de partidas históricas se basa en lo siguiente: se elige jugar contra, por ejemplo, Raúl Capablanca. Cuando le toca jugar a la IA, intentará localizar en la tabla de movimientos un FEN similar en el cual le tocase mover al citado jugador, y actuará imitándole. Si hubiera varias partidas que concordasen, elegiría una (aleatoriamente) entre aquéllas en las que el jugador histórico ganase.

Nótese que, para ambos motores, es muy probable que no encuentre ningún movimiento en su catálogo. Si esto ocurriera, ambos motores lanzarán una excepción que, controlada por el servidor, hará que éste ejecute otro motor de exploración distinto.

Capítulo 5

Inteligencia artifical

Como es inherente al problema del ajedrez desde el punto de vista computacional, y, dado que explorar todo el espacio de estados del juego no es computable, elegir la mejor jugada de la IA pasa por dos fases bien diferenciadas:

- Desarrollar el árbol de juego a un cierto nivel de profundidad acotado (ya sea con o sin poda).
- Del sub-árbol explorado, elegir el hijo a nivel 1 que arroje resultados más prometedores y tomar su camino (efectuar el movimiento).

Para el diseño del motor heurístico, nos hemos basado en un conjunto de heurísticas concretas y sencillas (en adelante, denominadas subheurísticas), que conforman una función de evaluación para cada tablero de la forma

$$H = a * h_1 + b * h_2 + c * h_3 + ... + n * h_n$$

siendo $a, b, c, ..., n$ constantes (en adelante, constantes de ponderación) y $h_1, h_2, h_3, ...h_n$ subheurísticas; esto es, una combinación lineal de heurísticas.

La clase Heuristic_Engine modela este comportamiento, permitiendo modificar o componer nuevos perfiles de inteligencia artificial cómodamente, desde una base de datos que almacena todos los parámetros necesarios.

Veamos una aproximación al código del motor heurístico:

```
public class Heuristic_Engine
{
	private ArrayList<Heuristic> heuristics;
```

```
        private ArrayList<Double> weights;

        [...]

        public double evaluate(Keyboard currentKeyboard,char colourToEvaluate)
        {
                double result=0;
                for(int i=0;i<this.heuristics.size();i++)
                {
                        result+=this.weights.get(i) *
                            this.heuristics.get(i).evaluate(currentKeyboard,colourToEvaluate);
                }
                return result;
        }

        [...]
}
```

Sea un breve resumen de algunas de las heurísticas:

random

Como su propio nombre indica, asigna a un tablero un valor totalmente aleatorio. Ha resultado ser inesperadamente útil, puesto que, especialmente en el medio juego (dependiendo de las heurísticas empleadas), es habitual que varias jugadas disintas tengan el mismo valor heurístico, con lo cual la máquina optaría por la primera de ellas.

La adición de esta heurística (con una constante de ponderación baja) elimina el factor de predictibilidad de la máquina.

static-matherial

Esta heurística asigna a cada pieza un cierto valor (parametrizado y configurable en la base de datos), invariable respecto al tiempo o a la posición del tablero. Es una idea primigenia enseñada a todos los principiantes de ajedrez. Hemos asignado los valores clásicos que empleó Aron Nimzowitsch en su tratado "Mi sistema"(TODO! cita), esto es, un punto al peón, tres puntos al caballo, tres puntos al alfil, 5 puntos a la torre, y diez puntos a la dama.

Ésta es, si bien un heurística intuitiva y útil, algo simplista, puesto que hay numerosas condiciones que modifican el valor intrínseco de una pieza en una partida:

- Se puede considerar que un caballo es un más valioso (debido a su movilidad) en las aperturas y comienzos del medio juego, pero menos valioso que un alfil al final del medio juego y en finales.
- Un peón pasado a punto de coronar es, desde luego, más valioso que un peón atrasado y bloqueado.
- En general, el factor movilidad de las piezas modifican su valor a lo largo de toda la partida.

open-columns

Las columnnas abiertas es un término acuñado y ampliamente estudiado por Aron Nimzowitsch en su obra de referencia, "Mi sistema". El temprano control de las mismas mediante una torre proporciona al bando que las controla la habilidad de lanzar ataques muy rápidos.

La implementación se limita a sumar 1.0 y 0.3 por cada columna abierta y semi-abierta ocupada por una torre o dama, respectivamente. Análogamente, resta el mismo valor para las columnas abiertas y semi-abiertas controladas por el bando contrario.

passed-pawns

Los peones pasados son un factor estratégico decisivo en el ajedrez, especialmente en la teoría de finales. No es en vano la cita "los peones son el alma del ajedrez". Si en una partida se diera una situación de peón pasado, la correcta defensa (y ataque o bloqueo) del mismo se convertirá en el centro estratégico de la partida, pues tiene la capacidad de desequilibrar totalmente el material ofensivo si llega a ser coronado, generalmente otorgando la victoria de este modo.

Esta heurística valorará con un +1 cada uno de nuestros peones pasados, y penalizará con un -1 a cada peón pasado enemigo. De este modo, se fomenta la creación de peones pasados, su perdurabilidad, y se intenta evitar la aparición de peones pasados en el enemigo.

pawn-dist-prom

Mediante esta heurística, intentamos minimizar la distancia de nuestros peones a la coronación. En su ausencia, el juego quedaría estancado. Imaginemos, por ejemplo, un final

de rey contra rey y peón en su fila original. Con las anteriores (y posteriores) heurísticas, absolutamente todo movimiento del bando en posesión del peón tendría un valor heurístico similar (salvo aquellos que desembocaran en la captura de nuestro peón).

Siendo así, dependeríamos del factor aleatorio para hacer avanzar nuestro peón.

El valor devuelto es el sumatorio de, para cada uno de nuestros peones, 7 - la distancia a la que están de coronar. (Siendo 7 el valor máximo para la distancia de coronación en un tablero de tamaño y configuración estándar).

pawn-chaining

Como ha sido comentado previamente, a pesar de su inaparente valor, los peones son el alma y el centro del ajedrez. Sin embargo, tanto en la apertura, medio juego y aun en los finales (mención aparte a los peones pasados, que perfectamente pueden ser válidos aun estando aislados), su poder radica fundamentalmente en sus compañeros de clase.

Mediante el valor devuelto por esta heurística, intentamos canalizar la partida hacia una correcta estructura de peones. Como ya ha sido mencionado, no es ésta una verdad intrínsecamente cierta por sí sola, pues hay excepciones como los peones pasados aislados en posiciones avanzadas o los peones en el centro.

El resultado retornado por esta subheurística consiste en el sumatorio de, para todos y cada uno de nuestros peones, la cantidad de peones que tiene a su derecha, ya sea a su lado, una casilla por delante o una casilla por detrás, es decir, el número de peones conectados entre sí.

number-of-movements

Representa una primera aproximación heurística a los factores de movilidad:

El valor heurístico devuelto se calcula de la siguiente manera: para cada una de nuestras piezas, sumamos la cantidad de movimientos válidos que pueden efectuar, devolviendo así el sumatorio. A diferencia de ideas anteriores, se ha demostrado empíricamente que no es conveniente penalizar el valor heurístico restándole el sumatorio de posibles movimientos del contrario, ya que, si así lo hiciéramos, la IA tendería constantemente a dar jaques improductivos contra el rey enemigo, estancando el juego, y, probablemente, empeorando la posición táctica en pos de un objetivo fútil.

dancing-queen

De nuevo, un concepto prontamente inculcado a los novicios en ajedrez, que, a base de experiencia, nos hemos visto forzados a inculcar a la IA: El temprano desarrollo de la dama no ocasiona sino una gran desventaja táctica, si bien hay líneas muy conocidas y jugadas por principiantes que lo llevan a cabo (gambito de dama aceptado, apertura Napoleón y apertura Patzer). El constante acoso por parte de piezas ligeras y aun peones del contrario nos forzarían a hacer numerosas jugadas de reubicación de la dama, situando al contrario muy por delante en la carrera por el desarrollo. Debido a lo anterior, el desarrollo de la dama antes del quinto turno es penalizado por esta heurística devolviendo un -1, o un 0 en caso contrario.

castling

Si bien ha sido una de las 'verdades universales' que más dudas hemos albergado a la hora de enseñar a nuestra IA, finalmente nos hemos decantado por su inclusión. Siguiendo los preceptos de esta heurística, si la máquina ha perdido la posibilidad de enroque tras no haberse enrocado (es decir, por haber movido el rey o las torres), penalizará la posición con un -1. Se podría haber ahondado más en la estructura de peones del enroque, pero hemos preferido dejar de lado esas disquisiciones, acaso para otra heurística distinta.

rook-queen-ending

Esta heurística, que modela el comportamiento en los finales de torre o reina, ha sido otro caso de desarrollo bajo demanda. En un modelo de computación ajedrecística como el presente (que jamás efectúa búsquedas orientadas, sino que explora subárboles y elige el hijo más prometedor), existe el problema de la idiosincrasia de los finales: Salvo que una cierta heurística determine lo contrario, tener al rey enemigo prácticamente encerrado en una fila o columna es tan bueno como cualquier otra posición. Siendo así, en el caso de ser un caso de final de rey y torre contra rey, la única esperanza de victoria de la IA consistiría en la exploración a una profundidad inmensa, inconmensurable si el humano oponente sabe cómo prolongar su (por otra parte, inexorable) final.

Así, la heurística va jugando con dos componentes:

- La distancia de Manhattan entre los dos reyes.

- La máxima dimensión del cuadrángulo en que nuestra torre o dama circunscribe al rey enemigo.

De este modo, se intentará acorralar al rey enemigo en un cuadrángulo lo menos posible, haciéndole retroceder mediante el apoyo de nuestro propio rey.

En cuanto al valor de cada una de las constantes de ponderación que acompañan a cada heurística, mucho se podría escribir:

Definitivamente, hay heurísticas de un valor o peso relativamente bajo, pero, ¿cuál serían los valores óptimos para las citadas constantes?

De nuevo, no es un problema decidible. Para hallar una aproximación razonablemente buena, consideramos una solución prometedora:

alggenetico

La construcción de un algoritmo genético para calcular valores sucesivamente mejores para las constantes de ponderación, tal que enfrentando a nuestra IA contra otro software de ajedrez (cuyo API ofrezca los rudimentos necesarios y un nivel de juego configurable tal que sea ligeramente superior a nuestro motor) en una cantidad suficiente de partidas, conseguiríamos la evolución del cromosoma (el conjunto de constantes de ponderación) hacia mejores resultados promedios.

Lamentablemente, y, si bien varias decisiones al respecto ya habían sido planeadas, carecemos de tiempo para llevar a cabo esta ampliación del proyeto.

En concreto, el gnuchess (que no es la interfaz gráfica proporcionada en sistemas linux, sino el mero motor subyacente) es sumamente interoperable, pudiendo un proceso comunicarse con él a través de stdin y stdout, en una notación cuasi-algebraica.

Otra idea complementaria a lo anterior que fue barajada fue la de utilizar un subconjunto de heurísticas u otro (con sendos distintos subconjuntos de constantes de ponderación) a lo largo de la partida, tal que en unos tableros que cumplieran ciertas condiciones se ejecutase un modelo heurístico determinado, y otros tableros, con distintas característicos, otro modelo. Sin embargo, teorizamos que sería sustituir un problema por otro nuevo, la estratificación de tableros en unas u otras características, por lo que optamos por la enseñanza a la IA de un conjunto de «verdades universales» aplicables para todo tablero, y ponderadas con las mismas constantes para todo tablero, pero con una ponderación que fuera siempre efectiva.

Asimismo, cuando inculcamos el conocimiento para ser capaz de efectuar finales de torre o dama, nos dimos cuenta que el modelo de computación utilizado hasta el momento (exploración y evaluación heurística con poda alfa-beta) era engorroso en determinados casos

(como en los citados finales). El cálculo de una heurística que determinase la victoria por mate en esos finales fue bastante complejo, donde un algoritmo determinista era casi trivial. En ese momento, planteamos el utilizar búsquedas orientadas en determinadas ocasiones en contra de exploraciones guiadas sólo por la evaluación heurística. Sin embargo, nos encontraríamos con el problema antes mencionado, la estratificación de los tableros según ciertas características, por lo que descartamos la idea, manteniendo un modelo computacional homogéneo a lo largo de toda la partida.

Capítulo 6

Paralelización

6.1. Introducción

En este capítulo se va a realizar un estudio sobre la paralelización del árbol de posibles movimientos generado por la IA.Esta utiliza un algoritmo MinMax iterativo para la toma de decisiones que es recorrido en anchura.

En la primera sección 6.2 **Entorno Experimental** se describe las distintas máquinas utilizadas y las herramientas software implementadas para llevar a cabo el estudio de paralelización.

En la sección de **?? Estrategias de paralelización** se explicará la tarea secuencial que se va a paralelizar y las distintas formas de paralelización llevadas a cabo.

Finalmente en la sección de 6.4 **Resultados** se analizará el impacto de la paralelización en tiempos de ejecución en el proyecto.

6.2. Entorno Experimental

El estudio se ha realizado utilizando el servidor Leviatán facilitado por el departamento de Arquitectura de la facultad de Físicas de la Universidad Complutense de Madrid. Un supercomputador de 48 Cores. También se ha utilizado un portátil Aspire 7741G de 4 cores, para contrastar datos en un equipo de bajas prestaciones. Las pruebas se han hecho utilizando 24 y 48 procesadores del supercomputador Leviatan y 4 en el portátil Aspire.

Se han incorporado varias herramientas software para ayudar al estudio de tiempos de la paralelización. 6.1 Un generador de tableros que se encargará de crear una lista de tableros y 6.2 una clase que se utilizará para realizar las pruebas de tiempos sin paralelizar.

Dicho generador de tableros juega partidas aleatorias y va guardando cada 3 movientos el tablero actual independientemente de que toque jugar a negras o blancas, de esta forma se tiene un número considerable de tableros en momentos distintos de la partida, pero sin ser una cantidad excesiva de datos. Sobre cada uno de estos tableros la IA genera un árbol con todos los posibles movimientos hasta dentro de 4 turnos(4 niveles), y aplicando una heurística determinada, calcula el mejor movimiento. Para el estudio de la paralelización se ha utilizado la heurística random". Esta heurística es la que menor coste computacional requiere,luego va a suponer la cota inferior respecto a obtener una ventaja con la paralelización.

Se ha medido varias veces el tiempo que tarda la IA en calcular el mejor movimiento para cada tablero, estos tiempos junto con el conteo del número de hojas se guardan en un fichero de salida que posteriormente se transladará a una tabla Excel. Basándose en estas tablas se ha llevado a cabo el estudio del SpeedUp en la sección 6.4 de resultados.

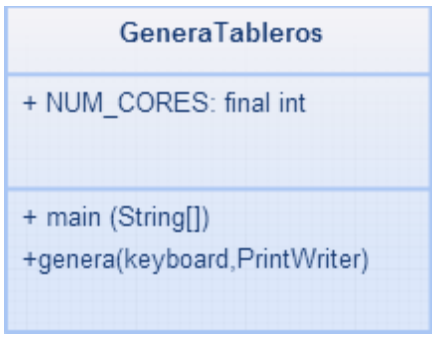


Figura 6.1: GeneraTableros

La clase 6.1 'generaTableros' crea un fichero con un tablero por línea, representados mediante su FEN. Parte, de el tablero inicial de una partida de ajedrez y juega movimientos aleatorios. Cada tres movimientos guarda el tablero resultante en el fichero, de esta forma hay una mayor variedad en los tableros almacenados. Se ejecuta varias veces de forma que haya suficientes partidas distintas. Este fichero será utilizado para realizar los cálculos en las distintas pruebas.

6.2'TestTiming' se ha utilizado para llevar a cabo las pruebas sin paralelizar. Lee del fichero de texto generado por 6.1 'generaTableros', con cada uno de estos tableros llama al método genera(keyboard,PrintWriter,int). El método genera recibe un tablero, un entero indicando el nivel de profundidad del árbol al que se quiere llegar y un fichero en el que se va a escribir el resultado (número de hojas generadas y tiempo que se tarda).

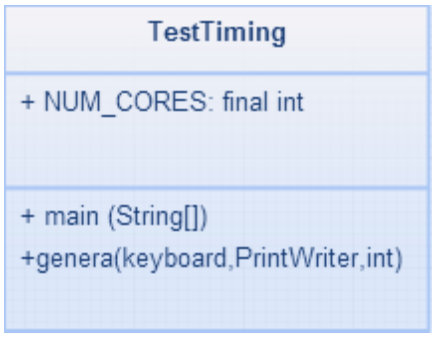


Figura 6.2: TestTiming

6.3. Estrategias de paralelización

6.3.1. Introducción

La tarea que se va a paralelizar es la creación del árbol de posibles movimientos por parte de la IA. La creación de dicho árbol de forma secuencial funciona del siguiente modo: A partir de un tablero inicial, mediante la clase 6.3 'Movementgenerator' se van a generar todos los posibles movimientos del siguiente turno. Para generar todos los posibles movimientos se van a calcular todos los posibles movimientos de cada pieza y concatenarlos en una lista. Sobre esta lista se aplicará una heurística determinada que dirá cual es el mejor movimiento que se puede jugar.

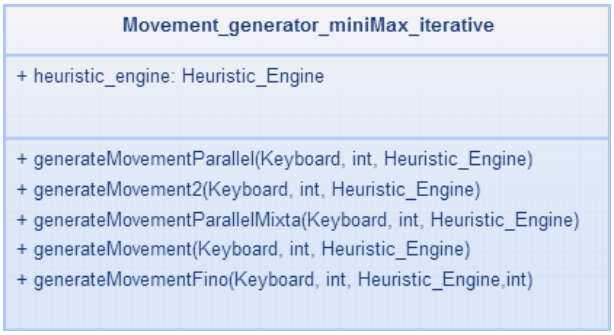


Figura 6.3: MovementGenerator

Esta clase 6.3 engloba generar el árbol de movimientos, y devolver el mejor movimiento posible dependiendo de la heurística especificada. El árbol de movimientos se genera mediante un recorrido en anchura y para hallar el mejor movimiento posible se utilza el algoritmo MinMax iterativo..

6.3.2. Interfaces de Java

La paralelización se ha llevabado a cabo implementando la interfaz Callable de Java. Y utilizando la interfaz Future y ExecutorService. Todas disponibles para versiones de Java iguales o superiores a la 1.5.

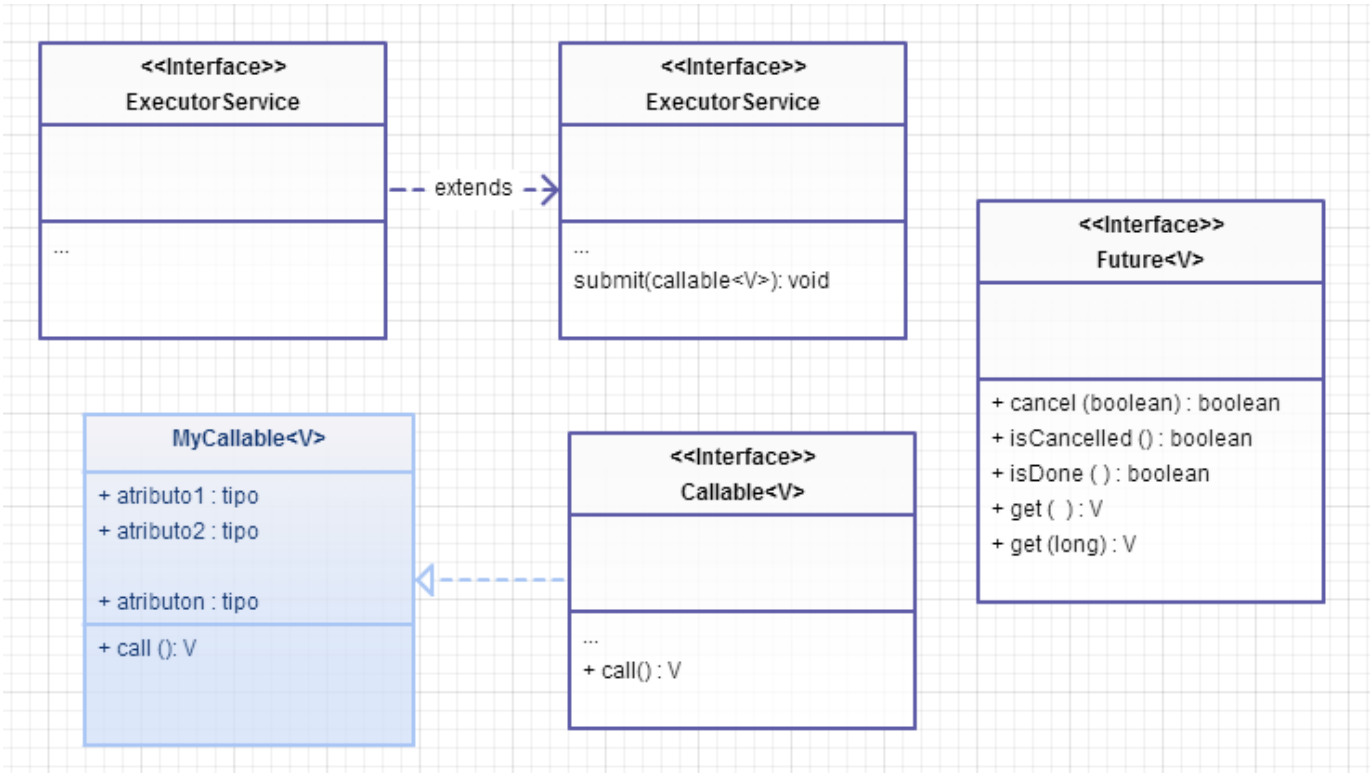


Figura 6.4: Interfaces de Java

- **Callable<V>:** Dispone de un método 'public V call()', donde 'V' es el tipo del objeto que nos interesa paralelizar. Es una interfaz similar a la interfaz Runnable, utilizada para la implementación de hilos. La ventaja de esta interfaz sobre Runnable es que nos permite interactuar con la interfaz Executor mediante el método call().
- **ExecutorService:** Interfaz que extiende la interfaz Executor. Permite crear un pool de hilos. Mediante el método *submit*(*callable*) se asignará a un hilo la tarea específica del método call() de la clase MyCallable. En caso de que todos los hilos estén ocupados, la tarea se encolará hasta que un hilo esté ocioso.
- **Future<V>:** La gran ventaja de esta interfaz es que te permite trabajar con futuros. Cada futuro va a guardar el resultado de un hilo cuando este termine su ejecución.

A continuación se incluye el pseudocódigo del funcionamiento de estas interfaces en el proyecto:

$executor = Executors.newFixedThreadPool(NUMCORES * 2)$;

Primero se crea el Pool de hilos que se van a utilizar. La mayor parte de los ordenadores actuales tiene Cores de doble núcleo, por lo que vamos a crear un Pool de hilos igual a la cantidad de Cores*2.

$ArrayList < Future < V >> result = newArrayList < Future < V >> ()$;

En la variable 'result' se van a guardar los valores futuros a medida que los hilos terminen.

$Callable < V > callable = newCallable("atributos")$;

En la clase 'MyCallable' que implementa la interfaz Callable<V>se va a especificar la tarea que se va a guardar en el valor futuro.

$Future < V > future = executor.submit(callable)$;

El valor de tipo 'V' devuelto por el método 'call()' se guarda en la variable future.

$result.add(future)$;

Ese valor se encola en nuestra lista de futuros.

El hilo principal podrá continuar su ejecución y sólo tendrá que esperar cuando necesite un valor futuro, y el hilo paralelo en cuestión no haya terminado aún.

6.3.3. Grano grueso

La primera forma de paralelización es a grano grueso. Cada hilo se va a encargar de devolver el mejor movimiento posible a partir de un tablero.

Partimos de un tablero inicial del que queremos hallar el árbol de posibles movimientos. Desplegamos el primer nivel del árbol de forma secuencial, y guardamos los tableros resultantes en una lista. La cantidad de tableros va a depender del momento de la partida en el que nos encontremos. Cada tablero de esa lista será trabajado por un hilo, el cual continuará desplegando el árbol de posibles movimientos hasta el nivel de profundidad deseado, para finalmente devolver el mejor tablero según la heurística utilizada.

Cuando todos los hilos en paralelo terminen. De forma secuencial, entre todos los tableros devueltos por la lista de tableros, se calculará el mejor movimiento posible aplicando la heurística deseada. Este mejor movimiento está representado por un tablero en formato FEN, que será el resultado que devolverá la IA como mejor movimiento posible.

6.5 y 6.6 son las clases principales encargadas de paralelizar a grano grueso. El objeto genérico <V>de la clase callable es en este caso *TupleKeyboardHeuristicValue*, una tupla

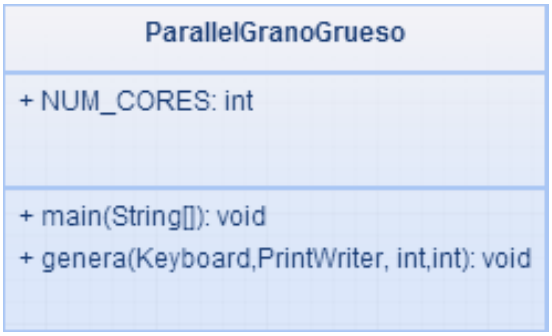


Figura 6.5: ParallelGranoGrueso

Figura 6.6: CallableGranoGrueso

del valor heurística y un tablero que indica el siguiente movimiento representado mediante su FEN.

6.3.4. Grano fino

En el caso de la paralelización a grano fino vamos a centrarnos en dividir la tarea en muchas tareas de poca carga. Cada hilo se va a encargar todos los posibles movimientos de una pieza.

Para hallar todos los posibles movimientos a partir de un tablero dado se calculan todos los posibles movimientos de cada una de las piezas. La paralelización a grano fino se va a encargar de paralelizar esta tarea. Cada hilo se encargará de calcular todos los posibles movimientos de una pieza y cuando termine se le asignará la siguiente pieza en la cola. Finalmente, todas las listas de posibles movientos de cada pieza se concatenarán formando la lista de posibles movimientos del tablero dado. Se seguirán evaluando los tableros hasta que se alcance el nivel de profundidad del árbol deseado. En este punto, de forma secuencial,

se buscará el tablero con mejor valor heurístico para devolver como resultado.

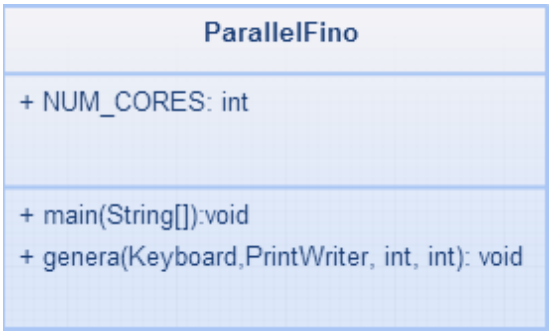


Figura 6.7: ParallelGranoFino

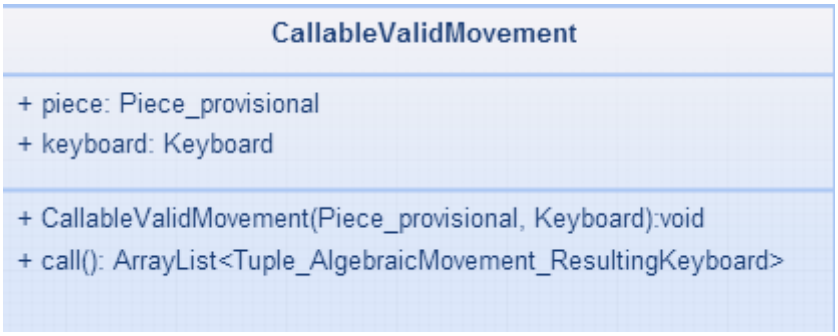


Figura 6.8: CallableGranoFino

En este caso el objeto <V>de la clase Callable es *ArrayList<TupleAlgebraicMovementResultingKeyboar* una lista de tuplas con el movimiento realizado y el tablero resultante tras realizar este movimiento. Cada hilo generado va a devolver una lista con todos los movimientos de una pieza.

6.3.5. Caso Mixto

En este caso vamos a integrar ambas formas de paralelización, tanto a grano fino como grueso. Va a haber un Pool de hilos encargados de calcular los posibles movimientos a partir de un tablero, y otros encargados de calcular todos los posibles movimientos de una pieza.

Partimos de un tablero inicial del que queremos hallar el árbol de posibles movimientos. Desplegamos el primer nivel del árbol de forma secuencial y guardamos los tableros resultantes en una lista. Cada hilo va a coger un tablero de la lista y hallar el mejor movimiento, generando el árbol de posibles movimientos hasta el nivel indicado. Al igual que en

la paralelización a grano grueso. Para calcular todos los posibles movimientos del árbol de movimientos cada hilo va a crear otro Pool de hilos por cada tablero.

En este nuevo Pool de hilos cada hilo se encargará de calcular todos los posibles movimientos de una pieza y cuando termine se le asignará la siguiente pieza en la cola. Al igual que la paralelización a grano fino.

6.4. Resultados

6.4.1. Estudio teórico

La ley de Amdahl permite averiguar la mejora máxima de un sistema cuando sólo una parte de este es mejorado. La fórmula de la Ley de Amdahl es la siguiente: $A = \frac{1}{1 - Fm + \frac{Fm}{Am}}$ siendo 'Fm':La fracción de tiempo que el sistema utiliza el subsistema mejorado y 'Am':el Factor de mejora introducido. El resultado, 'A', es la ganancia en velocidad consiguida en el sistema tras mejorar uno de sus subsistemas. La paralelización se base en la ley de Amdahl, ya que no se paraleliza todo el código, sólo una parte de este. Mediante la ley de Amdahl podemos calcular la mejora en velocidad que nos va a aportar la paralelización. A dicha mejora se le denomina Speed Up. podemos hallar el SpeedUp que se puede obtener al paralelizar mediante la ley de Amdahl utilizando: el número de procesadores(p) como factor de mejora 'Fm', y la fracción de tiempo que el sistema utiliza el sistema mejorado 'Am' es el cociente entre el tiempo de la parte paralelizable y el tiempo total de la aplicación (c). $SpeedUp - teorico = \frac{1}{1 - c + \frac{c}{p}}$ donde $c = \frac{Tp}{Tt}$. Es importante destacar que el SpeedUp-teórico es una cota superior, nos indica el Speed-Up máximo que se puede obtener con nuestra mejora. Y para su cálculo no se tiene en cuenta el tiempo consumido en creación,sincronización y comunicación de hilos por cada procesador, sumado a la competencia por rescursos, más la desigualdad en la carga de los procesadores.

El tiempo total de ejecución de la aplicación con el Aspire a un nivel de profundidad 3 es de $0,71s$, el tiempo de la parte que se puede paralelizar es de $0,61s$, por lo que $t = \frac{Tp}{Tt} = 0,87$. Luego tenemos un $SpeedUp - teorico = \frac{1}{1 - 0,87 + \frac{0,87}{p}}$ que va a depender del número de procesador que se utilicen. En nuestro caso, ese valor límite es desconocido ya que la cantidad máxima de procesadores utilizados han sido 48, y siempre se ha obtenido un mejora frente a utilizar un menor número.

6.4.2. Impacto de la paralelización en el tiempo de ejecución

En la figura 6.9 podemos observar el SpeedUp obtenido mediante la paralelización, es una pequeña extracción representativa de todos los tableros utilizados para realizar el cálculo.

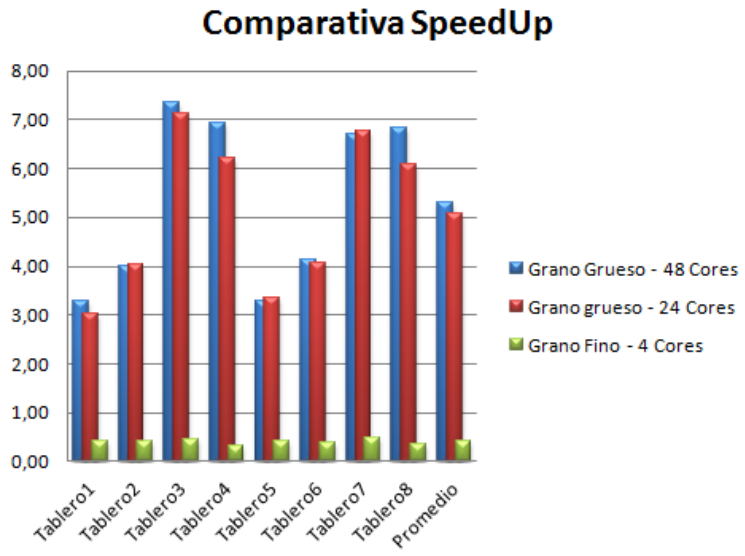


Figura 6.9: SpeedUp

- **Grano Grueso:** La paralelización a grano grueso implica la mayoría del código. Por lo que el $SpeedUp - teorico = \dfrac{1}{1 - 0,87 + \dfrac{0,87}{p}}$. Con los datos obtenidos en la figura 6.9 podemos observar un SpeedUp final de 5,39 y 5,04 utilizando 48 y 24 Cores en el leviatan respectivamente. Siendo el SpeedUp-teórico 6,75 y 6,01. El SpeedUp-teórico establece la cota superior sin tener en cuenta el tiempo extra para incorporar la paralelización, se concluye que el resultado es muy satisfactorio.

- **Grano Fino:** La parte a paralelizar a grano fino es la parte con más carga de trabajo, por lo que el SpeedUp-teórico sigue siendo el mismo:$SpeedUp - teorico = \dfrac{1}{1 - 0,87 + \dfrac{0,87}{p}}$. El tiempo que tarda cada hilo en calcular todos los movientos de una pieza es del orden de 10^{-7}. A pesar de tener un SpeedUp-teórico de 6,75 para 48 procesadores, la nimia cantidad de carga de cada hilo va a provocar que sea más costoso añadir el código para paralelizar(generar la tarea y asignarla a un valor futuro) que el tiempo ganado en desarollar la tarea en paralelo.

Como se puede comprobar con los datos calculados en la 6.13 en muy pocos casos se obtiene un tiempo mejor. No sólo eso, sino que en la mayoría se obtendrán tiempos peores a los de 6.10 el código de forma secuencial .

- **Recolector de basura:** Este proyecto ha sido desarrollado en Java, que utiliza un recolector de basura automático. Este utiliza varios hilos trabajando en paralelelo con la carga distribuida entre los procesadores. Se ha estudiado el funcionamiento de este limitando el número de hilos a 1, de esta forma los procesadores pueden dedicar más carga de trabajo a la generación del árbol de movimientos. En la figura 6.14 se puede ver los datos obtenidos en el Leviatan con 48 procesadores y 1 sólo hilo para el recolecctor de basura. Comparando esos datos con los del código en secuencial 6.10 observamos que el tiempo obtenido limitando el número de hilos a 1 es casi el doble que utilizando todos los hilos, tarda 1,72 veces más, lo que indica que es ventajoso utilizar todos los hilos del recolector de basura por defecto establecidos.

6.5. Conclusiones

Se ha obtenido un rendimiento muy favorable realizando la paralelización a grano grueso. La paralelización a grano fino no mejora los tiempos de ejecución de la aplicación, a pesar de ello, se ha incorporado al proyecto ya que futuras versiones de este podrían hacer viable su uso. La paralelización Mixta mejora el rendimiento en menor medida que la paralelización a grano grueso por la ineficiencia de la paralelización a grano fino.

6.6. Tablas de datos

Las tablas utilizadas en esta sección son una pequeña extracción. El tiempo de cada tablero ha sido calculado varias veces, los datos aquí representados son una media de esos tiempos.

TABLERO	Nº DE HOJAS	TIEMPO(s)
1	197281	2,677158216
2	221030	5,014734164
3	409965	15,46243226
4	277989	9,006662346
5	221030	4,75301339
6	409965	8,008653004
7	277989	9,297660874
8	298774	12,64580189
9	1114601	82,71700959
10	72014	7,179124905
11	369877	16,29570196
12	444451	20,22461212
13	396886	11,64820243
14	639475	27,8174347
Promedio	382237,6429	17,62839384

Figura 6.10: secuencial 48 procesadores

TABLERO	Nº DE HOJAS	TIEMPO(s)
1	197281	0,899885115
2	221030	1,685624929
3	409965	5,197456221
4	277989	3,027449528
5	221030	1,59765156
6	409965	2,691984203
7	277989	3,125264159
8	298774	4,250689712
9	1114601	27,80403684
10	72014	2,413151229
11	369877	5,477546879
12	444451	6,798188949
13	396886	3,915362161
14	639475	9,35039822
Promedio	382237,6429	5,925510534

Figura 6.11: Leviatan 24 procesadores grano grueso

TABLERO	Nº DE HOJAS	TIEMPO(s)
1	197281	0,787399475
2	221030	1,474921813
3	409965	4,547774193
4	277989	2,649018337
5	221030	1,397945115
6	409965	2,355486178
7	277989	2,73460614
8	298774	3,719353498
9	1114601	24,32853223
10	72014	2,111507325
11	369877	4,792853519
12	444451	5,948415331
13	396886	3,425941891
14	639475	8,181598443
Promedio	382237,6429	5,184821717

Figura 6.12: Leviatan 48 procesadores grano grueso

TABLERO	Nº DE HOJAS	TIEMPO(s)
1	197281	2,944874037
2	221030	5,516207581
3	409965	17,00867548
4	277989	9,90732858
5	221030	5,228314729
6	409965	8,809518304
7	277989	10,22742696
8	298774	13,91038208
9	1114601	90,98871055
10	72014	7,897037396
11	369877	17,92527216
12	444451	22,24707334
13	396886	12,81302267
14	639475	30,59917817
Promedio	382237,6429	19,39123322

Figura 6.13: Leviatan 48 procesadores grano fino

TABLERO	Nº DE HOJAS	TIEMPO(s)
1	197281	4,551168967
2	221030	8,525048079
3	409965	26,28613484
4	277989	15,31132599
5	221030	8,080122763
6	409965	13,61471011
7	277989	15,80602349
8	298774	21,49786322
9	1114601	140,6189163
10	72014	12,20451234
11	369877	27,70269334
12	444451	34,38184061
13	396886	19,80194413
14	639475	47,289639
Promedio	382237,6429	29,96826953

Figura 6.14: Recolector de basura con 1 hilo

Capítulo 7

Conclusiones

Debido fundamentalmente a la magnitud del proyecto, y a pesar de haber tenido un diseño claramente definido desde el principio, las necesidades del mismo han ido creciendo por sí mismas, en ocasiones demandando bastante código para la mera instrumentalización del mismo, como por ejemplo, para poder depurar y calibrar correctamente las evaluaciones heurísticas.

De este modo, y bajo demanda, en ocasiones hemos descubierto pequeñas imprecisiones en el diseño inicial en etapas avanzadas, careciendo de tiempo y recursos para hacer un refactoring óptimo. Asimismo, ha habido detalles que, a pesar de haber sido planteados o aun diseñados en un momento determinado, no hemos podido llevar a cabo:

- Optimización de la generación de los hijos a nivel 1
  Mediante la reescritura parcial de una sección determinada de este algoritmo nos evitaríamos la masiva y presumiblemente innecesaria invocación al constructor de copia de Piece.
- Algoritmo genético de aprendizaje
  Como ha sido indicado previamente **??** , nos hubiera gustado poder terminar un algoritmo genético que devolviera una ponderación de heurísticas que arroje buenos resultados.
- Traducción de un subconjunto del sistema a C++
  La traducción del motor en su conjunto (esto es, la generación de hijos a nivel 1, la generación de hijos a nivel N y el motor heurístico de evaluación), invocado a través

de JNA, dotaría al sistema global de una eficiencia superior, permitiéndole explorar a profundidades mayores.

- Sistema distribuido
  Asimismo, se ha diseñado el germen de una arquitectura cliente-servidor de un sistema de computación distribuido basado en master-slave. De nuevo, nos ha sido imposible llevarlo a cabo por falta de tiempo y recursos. Este punto presenta un dificultad adicional inherente al ajedrez, y es la replicación de tableros en distintas ramas a la hora de explorar un subárbol de juego. Para evitar la expansión y evaluación en tableros repetidos a lo largo y ancho del sistema distribuido pensamos en varias estrategias, cada una de las cuales conlleva un compromiso:

  - Mantener un sistema de paso de mensajes entre los nodos de computación y el master centralizado supone dificultades en el sincronismo, así como una considerable dedicación de los recursos de la máquina a la sincronización en vez de al mero cómputo.
  - Permitir a los nodos de cómputo replicar los tableros libremente ahorra tiempos de sicronización (ya fuera ésta mediante consulta a una caché de tableros centralizada o a través de paso de mensajes), pero probablemente fuera más lento que intentar evitar la innecesaria carga de trabajo.
  - Sea, sin haber tomado benchmarks de tiempos, una aproximación ligera pero prometedora: evitar la repetición de cómputo entre distintos nodos de computación, si y sólo si el tablero cuya replicación se pretende evitar tiene una profundidad en el subárbol menos que un determinado valor i. De este modo, evitamos que un mismo tablero sea expandido en dos nodos de cómputo si del tablero cuelga un subárbol de tamaño considerable, evitando el impacto debido al sincronismo en aquellos nodos cercanos a las hojas, en los que el tiempo de expansión y consecuente evaluación podría ser despreciable, aun siendo tableros ya explorados previamente.

- Motor de exploración de partidas históricas
  Aunque se avanzó mucho en lo relativo a este motor, no hemos podido disponer de tiempo para terminarlo. A día de hoy, existe un traductor tal que, proporcionándole como entrada la ruta de un fichero en formato PGN, importa éste a nuestro sistema

de base de datos, populando así el catálogo del sistema. Una vez se contase con esa información, la generación del motor en sí mismo sería relativamente sencilla, limitándose a ejecutar un par de transacciones sobre la base de datos.

En general, una de las cosas de las que «nos arrepentimos» (si bien todo error conlleva un aprendizaje) es el no haber delegado la función de, dado un tablero determinado generar sus hijos a nivel 1; al motor gnuChess, en cuya eficiencia podemos confiar. De este modo, hubiéramos contado con tiempo para profundizar mucho más en los motores de exploración, la inteligencia artficial, el paralelismo y el cómputo distribuido.

Apéndice A

Manual de uso del aplicativo gráfico

En la figura A.1 se puede apreciar el aspecto visual de la aplicación.
Las acciones habilitadas para el usuario son las siguientes:

Conectar

Permite al usuario autentificarse contra la base de datos como usuario registrado para poder comenzar a utilizar la aplicación.

Desafiar

Permite al usuario, una vez logueado, desafiar a otro usuario. Para ello es requisito indispensable que ambos usuarios no tengan ya una partida abierta entre ambos.

Esta funcionalidad se puede disparar clicando, en el panel derecho, sobre el usuario a quien se quiere desafiar. Este desafío puede ser aceptado o rechazado (aunque aquellos jugadores que representan un cierto perfil de inteligencia artificial aceptarán siempre).

Rendirse

Permite a un usuario rendirse en una cierta partida, es decir retirarse (perdiendo) sea cual sea la situación, siempre y cuando no haya finalizado ya.

Salir

Cierra la aplicación desconectando al usuario.

Realizar un movimiento

Para realizar un movimiento en una partida abierta basta con clicar en la pieza que deseamos mover y después volver a clicar sobre la casilla destino del movimiento.

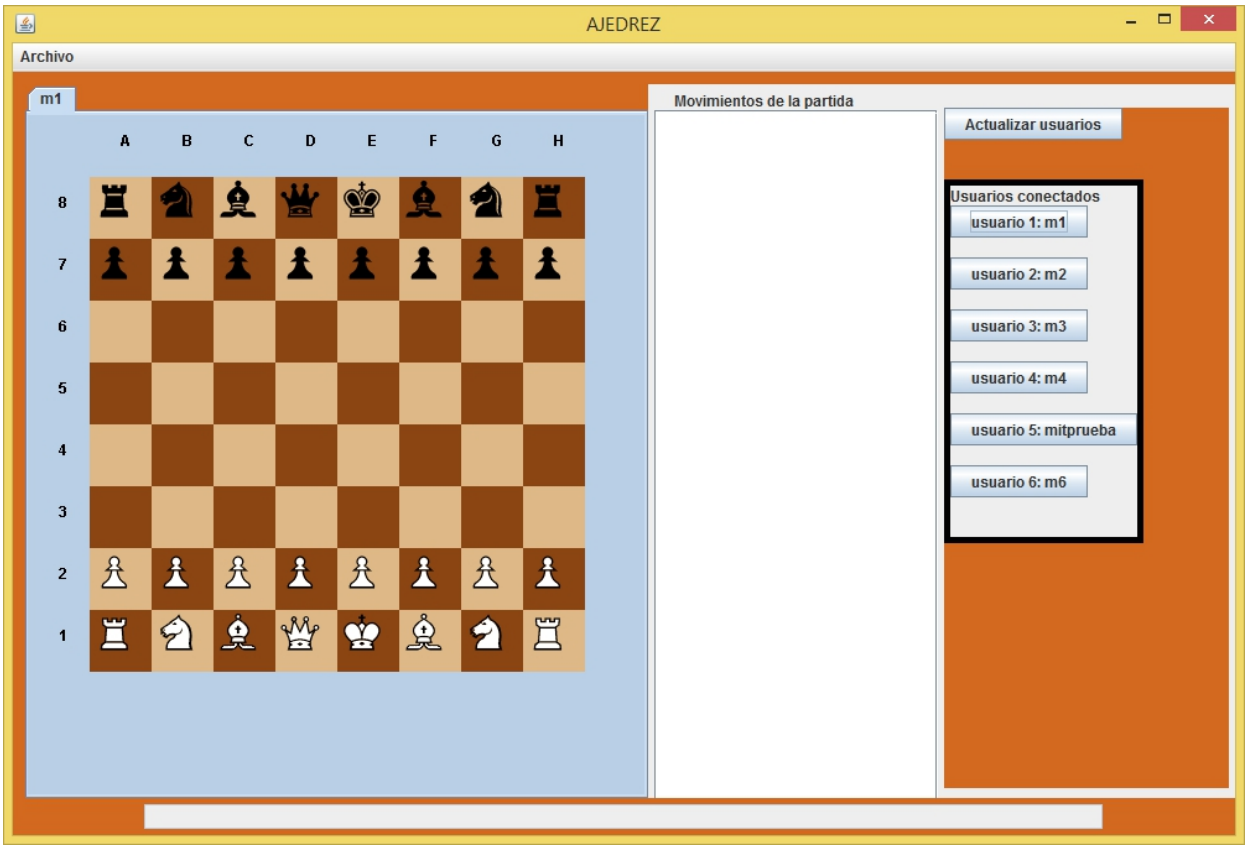


Figura A.1: Aspecto gráfico de la aplicación

Apéndice B

Autoría del presente documento

Glosario de términos: Daniel González Marcos.

Capítulo 1, Introducción: Daniel González Marcos.

Capítulo 2, Trabajos anteriores y estado del arte: Daniel González Marcos.

Capítulo 3, Arquitectura de comunicación: Daniel González Marcos.

Capítulo 4, Motor de juego: Jorge Pérez Barrio.

Capítulo 5, Inteligencia artificial: Daniel González Marcos.

Capítulo 6, Paralelización: Eduardo Bouza Cubo.

Capítulo 6.5, Conclusiones: Daniel González Marcos.

Apéndice A, Manual de uso: Jorge Pérez Barrio.

:

Bibliografía

[1] http://en.wikibooks.org/wiki/LaTeX/, *Referencia online de LaTeX.*

[2] http://apronus.com/chess/wbeditor.php *Configurador de tableros,generador de PGN y FEN.*

[3] http://www.pictureresize.org/online-images-converter.html *Conversor de imágenes online.*

[4] http://www.gnu.org/software/chess/manual/ *Página del proyecto gnuChess.*

[5] Aron Nimzowitsch, *Mi sistema.* Editorial Fundamentos, 8º edición, 1963.

[6] A. P. Sokolsky, *Celadas en ajedrez.* Ediciones limitadas Catalán, 1976.

[7] A. P. Sokolsky, *Aperturas abiertas.* Ediciones limitadas Catalán, 1967.

[8] A. P. Sokolsky, *Aperturas semi-abiertas.* Ediciones limitadas Catalán, 1967.

[9] A. P. Sokolsky, *Aperturas cerradas.* Ediciones limitadas Catalán, 1967.

[10] https://chessprogramming.wikispaces.com *Comunidad online de inteligencia artificial en ajedrez.*