Proyecto de Grado 2013

Desarrollo y Optimización de Motores de Ajedrez

Lilian Cazalás, José Artola

Facultad de Ingeniería - Universidad de la República
Montevideo, Uruguay

Área temática: Programación y Ajedrez
Supervisor: Dr. Ing. Pablo Romero
Usuario responsable: Ing. Gonzalo Varalla

"Un juego es una situación conflictiva en la que uno debe tomar una decisión sabiendo que los demás también toman decisiones, y que el resultado del conflicto se determina, de algún modo, a partir de todas las decisiones realizadas."

John von Neumann

Reconocimientos

A nuestros tutores Gonzalo Varralla y Pablo Romero, por su apoyo, entusiasmo, conocimiento, y pasión por el ajedrez. Gonzalo, con sus asignaciones que marcaron nuestro camino, y por recordarnos siempre la diferencia entre piezas y fichas. Y Pablo, por su preocupación, coordinación, disponibilidad en todo momento y gran aporte en la optimización final de algoritmos.

A Mathías Oliveri y Andreas Fast por su herramienta que nos ayudó en el análisis de las partidas jugadas en el último torneo y nos permitió demostrar que logramos el objetivo del nuestro proyecto.

A Sergio Nesmachnow, Jorge Triñanes y Carlos Testuri, por dedicarnos el tiempo necesario para formar el tribunal evaluador.

Queremos agradecer también a nuestras familias, que nos acompañaron a lo largo de la carrera y siempre estuvieron para alentarnos en los momentos difíciles.

Seguramente nos faltarán palabras para expresar como hubiéramos deseado, lo que ha significado para nuestro proyecto y para nuestras personas esa indeclinable confianza en nuestro quehacer.

Resumen

Los motores de ajedrez han evolucionado a pasos agigantados desde sus inicios teóricos en la mente de varios científicos, como una idea casi imposible de llevar a cabo con los recursos disponibles de la época; hasta la actualidad, donde tanto los aficionados como los grandes maestros y campeones de ajedrez los utilizan como oráculos para mejorar sus técnicas y estrategias.

Si bien en un principio, el problema de construir un motor de ajedrez no tenía importancia práctica, sí había un gran interés teórico dado que la solución encontrada a este problema sería el puntapié inicial para atacar otros problemas de naturaleza similar pero de mayor importancia. Dada su complejidad y riqueza combinatoria, este juego-ciencia ha cautivado la atención de pioneros en el desarrollo de las Telecomunicaciones y la Teoría Computacional como ser Claude Shannon [1] y Alan Turing [2], quienes fueron los primeros en dar propuestas para el desarrollo de autómatas de ajedrez. Otras causas que han motivado la creación de motores de ajedrez son: el entretenimiento propio (permitiendo que los jugadores practiquen y se diviertan sin necesidad de convocar oponentes humanos), poder utilizarlo como herramienta o soporte de análisis, y para investigación del conocimiento humano. Como se puede ver, este juego es centro de atención tanto en inteligencia artificial como en la psicología cognitiva del ser humano.

El funcionamiento de los programas de ajedrez consiste, esencialmente, en explorar un número muy elevado de posibles movimientos futuros y aplicarles una función de evaluación al resultado. Nuestro trabajo, busca modestamente recorrer el camino necesario para la construcción y optimización de un motor de ajedrez. Tomaremos como base un motor existente, estudiaremos primero su estructura interna de datos y funciones básicas de trabajo, para luego, en base a lo investigado desarrollar una función de evaluación completa, y dentro de nuestras posibilidades superar el desempeño del motor original.

Para la selección del motor a mejorar, en primera instancia, realizamos una búsqueda de motores de código abierto para estudiarlos y ver cuál se adaptaba más a nuestras necesidades. Después de elegido el motor original

llamado “Winglet” [3], realizamos diferentes asignaciones para familiarizarnos con éste e ir creando funciones auxiliares que nos ayudarían más adelante, al momento de crear nuestra función de evaluación de jugadas.

Luego de haber implementado mejoras y una nueva función de evaluación de posiciones surgió el primer motor de Facultad de Ingeniería, al que hemos llamado **“Joli”**. Joli se implementó en lenguaje C++ y permite ser ejecutado por línea de comandos o utilizando una interfaz gráfica como Winboard [4]. Posee un archivo de configuración el cual permite parametrizar variables tales como tiempo de análisis, cantidad de jugadas futuras a considerar y diferentes configuraciones de evaluación para dar más importancia a determinadas métricas de evaluación por sobre otras. Realizamos un conjunto de pruebas de control, y con muchas expectativas entablamos diferentes torneos entre el motor original, Winglet, y distintas configuraciones de Joli. Para nuestra satisfacción, en el último torneo “triangular” jugado, Winglet vs Joli (en dos configuraciones distintas) ambas versiones de Joli salieron sustancialmente mejor posicionadas que Winglet.

Por último, para dar un cierre lo más objetivo posible, y ayudados por el proyecto “Medición de fuerza de juego por comparación de rendimiento con motores” que procesaron las partidas realizadas, evaluamos el nivel de juego de nuestro motor en comparación con su versión original cumpliendo el objetivo principal del proyecto que consistía en el desarrollo y la optimización de un motor existente.

Índice general

Capítulo 1

Introducción

El proyecto actual, intenta recorrer los pasos necesarios para desarrollar y optimizar un motor de ajedrez, partiendo de un motor base a elección e interiorizarnos en cada aspecto de su construcción. Comenzaremos con un relevamiento de la historia del Ajedrez, describiendo sus orígenes y evolución, así como también haremos un breve repaso de las máquinas y personas que de una u otra forma fueron marcando el camino para llegar al día de hoy, donde ya desde hace un tiempo, el mejor jugador del mundo no tiene sangre corriendo por sus venas.

En este documento, iremos describiendo todas las etapas recorridas a lo largo de estos ocho meses de trabajo, dividiendo cada etapa en capítulos, de forma tal de agrupar el contenido de una forma amena. En el siguiente capítulo daremos un marco histórico en el que recorreremos los orígenes del juego por el siglo VII, su difusión desde Asia a Europa y su evolución durante los siglos XV y XVI, donde el Ajedrez obtendrá el catálogo de piezas y movimientos que ha conservado hasta el día de hoy. También en este capítulo, y para proveer un marco local, haremos un breve repaso de los emprendimientos que está llevando a cabo la Universidad de la República, con sus diferentes Facultades en torno a este juego-ciencia.

El tercer capítulo hará un breve recuento histórico de los motores de Ajedrez, desde sus inicios con autómatas mecánicos hasta la victoria de Deep Blue en el año 1997. Veremos también, de forma resumida, algunas pautas necesarias a tener en cuenta en el diseño de motores de Ajedrez y cerrando el capítulo haremos una pequeña revisión de los motores actuales.

Es en el Capítulo 4 en donde detallaremos cada una de las etapas del desarrollo de nuestro motor. Comenzaremos por el relevamiento de información sobre la construcción de motores. Detallaremos un conjunto de definiciones, necesarias para comprender mejor las siguientes secciones del capítulo, como por ejemplo: las etapas del juego, los diferentes tipos de partidas, las notaciones utilizadas para el registro de éstas y dos temas no menos importantes: cómo representar un tablero y la comunicación del motor con la interfaz gráfica. Luego, dedicaremos una sección completa al proceso de selección del motor y detallaremos las características del motor implementado, su representación interna del tablero, la estructura de los movimientos y su generador. Otra sección será dedicada a las asignaciones de programación indicadas por los tutores y finalizando el capítulo, describiremos los contratiempos sufridos, las pruebas realizadas y el proceso de optimización del motor.

El Capítulo 5 estará dedicado completamente al análisis de los resultados obtenidos durante los torneos finales, en los que enfrentamos dos instancias de nuestro motor, con configuraciones que consideramos óptimas luego de analizar diferentes ponderaciones en la etapa anterior, contra el motor elegido como base. Con estos resultados y apoyados por la herramienta provista por el proyecto de grado *"Medición de fuerza de juego por comparación de rendimiento con motores"* analizaremos las partidas jugadas en este torneo para determinar si logramos el objetivo de mejorar el motor tomado como "motor base" y poder ver la magnitud de la mejora.

El siguiente capítulo lo dedicaremos exclusivamente a un resumen de posibles trabajos a futuro y mejoras a realizar sobre nuestro motor "Joli" para lograr mayor calidad de juego y optimizar su desempeño. Estas mejoras se centrarán en dos aspectos fundamentales, mejorar la función de evaluación mediante diferentes técnicas y estudiar diferentes métodos para incrementar la velocidad de análisis y así reducir el tiempo para analizar las jugadas a cierta profundidad ó alcanzar mayor profundidad en el mismo tiempo.

Luego de todo este camino, consideramos importante dedicar un capítulo para plasmar nuestras conclusiones finales y dar un cierre correcto al proyecto.

Capítulo 2

Ajedrez

2.1. Historia y evolución del juego

El juego de ajedrez posee una historia de más de mil años, su origen se remonta al siglo VII en India como una variante del Chaturanga o Juego del Ejército [6]. Gracias al comercio que estos pueblos mantenían con sus vecinos Persas, se difundió al imperio Bizantino y posteriormente a toda Asia. Su expansión a Europa llegó de la mano de los musulmanes entre los años 700 y 900 de nuestra era cuando España pasó a estar bajo el dominio del Islam. También se tienen registros de versiones antiguas de ajedrez jugadas por Vikingos y Cruzados que regresaban de Tierra Santa. Durante la Edad Media, en España e Italia fue donde más se practicó la disciplina basada en las normas Árabes de juego.

La era moderna del Ajedrez se remonta a los siglos XV y XVI, donde las piezas obtuvieron la forma que tienen al día de hoy, junto con sus reglas y movimientos; a los peones se los habilita a mover dos casillas en el primer movimiento y a ser coronados por otra pieza al llegar al otro extremo del tablero.

Un momento decisivo para la unificación de las reglas se produjo en 1450 con la invención de la imprenta, la cual permitió imprimir libros de ajedrez con reglas y ensayos, pudiendo ser distribuidos de forma más eficiente. En 1737 Philip Stamma de Aleppo publica en París el libro *"El noble juego del ajedrez"* [7] donde se utiliza por primera vez en la historia un nuevo sistema de notación para representar los movimientos de las piezas llamado "Nota-

ción Stamma"; que posteriormente se renombró a "Notación algebraica" [8] y es actualmente recomendado por la FIDE [9] (Federación Internacional de Ajedrez). La notación algebraica (ver sección 4.1.3 pág 26) consiste en una representación escrita de los movimientos realizados durante una partida, permitiendo documentar partidas completas (cada escaque del tablero está identificado por un par de coordenadas).

2.2. Ajedrez en el Uruguay

En Uruguay tenemos dos Asociaciones que se dedican a la práctica del deporte: la *Federación Uruguaya de Ajedrez* y la *Asociación de Ajedrecistas del Uruguay*, las cuales nuclean un gran número de clubes tanto de la capital, como del interior del país. La *Federación Uruguaya de Ajedrez* (FUA), fundada en el año 1926, es el organismo rector del Ajedrez en Uruguay, tiene su inscripción en la *Federación Internacional de Ajedrez* (FIDE) y en el *Ministerio de Deporte y Cultura del Uruguay*. Además cuenta con el apoyo del *Servicio Central de Bienestar Universitario* (SCBU), centro de los servicios sociales de la Universidad de la República (UdelaR), cuyas actividades principales se cumplen en las áreas de salud, becas, cultura, deportes, alimentación y recreación teniendo como principal objetivo mejorar la calidad de vida de trabajadores/as y estudiantes.

Actualmente se llevan a cabo diferentes proyectos que involucran al Ajedrez en un marco estudiantil multidisciplinario apoyado por *Bienestar Universitario*, la *Universidad de la República* con sus diferentes Facultades y la *Federación Uruguaya de Ajedrez*. Existen talleres itinerantes en las diferentes Facultades donde se dictan clases para todos aquellos interesados en aprender la disciplina y se reúnen estudiantes para practicar el deporte.

Dentro de este emprendimiento en torno al Ajedrez, se han planteado diferentes enfoques, desde el punto de vista técnico, como es nuestro proyecto, cuyo objetivo es desarrollar y optimizar un motor existente, y su proyecto "hermano": Evolución del Rendimiento en Ajedrez, el cual aportará su valoración de jugadas a nuestras partidas y las de cualquier otro jugador que las comparta. Ambos proyectos son de gran utilidad en la práctica para los entrenadores uruguayos.

El día 30 de Octubre de 2013, el Proyecto *"Ajedrez UdelaR"* perteneciente al Área de Cultura de Bienestar Universitario, realizó un Seminario [10] en el cual presentamos nuestro motor junto con otros trabajos académicos, enmarcados en la órbita de la Universidad de la República, los cuales realizan diferentes estudios, tomando como base el juego de Ajedrez. El Seminario fue seguido por videoconferencia desde *Regional Norte* en el departamento de Salto y desde la *Universidad de Vigo* [11], por las máximas autoridades de esa casa de estudios europea, junto a investigadores interesados en trabajar en conjunto con nuestros académicos. Con la Universidad de Vigo se estudia firmar un convenio para trabajar en conjunto en este tipo de iniciativas. La iniciativa recibió el respaldo del Rectorado de la Udelar, a través de la firma de un Convenio.

Figura 2.1: Seminario Ajedrez 2013

Los trabajos presentados fueron:

- **"Impacto Cognitivo Ajedrez en la Escuela"**, a cargo de las Psic. Mag. Karen Moreira y Psic. Mag. Karina Curione del Instituto de Fundamentos y Métodos de la Facultad de Psicología

  Donde se presentó un informe sobre los estudios del efecto que produce a nivel cognitivo este juego en niños en etapa escolar y los posibles beneficios que aporta practicar esta disciplina desde temprana edad

- **"Ajedrez y género"**, presentando primeras hipótesis y pesquisas de campo los sociólogos Dinorah Rossano, Bruno Andreoli y Agustina Marques de la Facultad de Ciencias Sociales. Se presentó un estudio de cuánto depende el gusto y la disposición a jugar Ajedrez según el género de la persona

- **"Desarrollo de motor de ajedrez"**, presentado por nosotros (José Artola y Lilian Cazalás) bajo la tutoría del Doctor, Ingeniero Pablo Romero y el Ingeniero Gonzalo Varalla de la Facultad de Ingeniería

- **"Medición de fuerza de juego por comparación de rendimiento con motores"** presentado por Andreas Fast y Mathías Oliveri, bajo la tutoría del Doctor, Ingeniero Pablo Romero y el Ingeniero Gonzalo Varalla de la Facultad de Ingeniería

  Proyecto desarrollado en paralelo con el nuestro, el cual fue de mucha utilidad para nosotros dado que su producto final ayuda a un ajedrecista (o motor) a evaluar su nivel de juego en partidas o torneos comparando sus decisiones con las de otro motor oráculo.

- **"Aplicación del procesamiento de imágenes para el seguimiento de una partida de ajedrez"**, presentado por Cedric Zoppolo y el Ingeniero Alvaro Gomez de la Facultad de Ingeniería, Instituto de Ingeniería Eléctrica.

  El proyecto consiste en el seguimiento de partidas de ajedrez –y transmisión on line- a partir del procesamiento de imágenes. Algunos expertos en ajedrez entienden que esta propuesta podría llevar a modificar el modo de transmisión de partidas a nivel mundial

- **"Ajedrez y diseño"**, presentado por el Licenciado Daniell Flain de la Facultad de Arquitectura, Escuela Universitaria Centro de Diseño.

  Desde el punto de vista de la estructura y forma de los tableros y piezas, se presentaron diferentes trabajos de diseños en base a diferentes corrientes artísticas.

Capítulo 3

Motores de Ajedrez

3.1. Historia y evolución de motores

El siglo XVIII es fundamental para comprender el mundo moderno, pues muchos de los acontecimientos políticos, sociales, económicos, culturales e intelectuales del siglo han extendido su influencia hasta la actualidad, como ser la revolución industrial, la creación de la máquina a vapor, el desarrollo del cálculo infinitesimal, la creación de ramas de la matemática como la topología y el cálculo complejo. Durante este movimiento intelectual conocido como Ilustración surgió la idea de crear una máquina (o motor) capaz de jugar al ajedrez. Algunas de las causas que han motivado la construcción de motores de ajedrez son: el entretenimiento propio, pudiendo practicar el deporte sin necesidad de convocar más gente; utilizarlo como herramienta o soporte de análisis, para competiciones entre computadoras de ajedrez, y como investigación del conocimiento humano, dado que se trata de un juego de estrategia y razonamientos a futuro.

En **1769** se presentó un autómata que jugaba al ajedrez llamado "El Turco" [12], construido por Wolfgang Von Kempelen. Estaba constituido por una caja de madera de 1,20 m de largo por 60 cm de profundidad por 90 cm de alto y un maniquí que "realizaba los movimientos". En su interior se podía ver un mecanismo de relojería que al activarse era capaz de jugar una partida de ajedrez contra un jugador humano ganando la mayoría de las partidas. Durante muchos años realizó giras por el mundo enfrentándose entre otros con Napoleón Bonaparte y Benjamin Franklin. Si bien se continuaba exhi-

biendo con gran éxito como una máquina que jugaba al ajedrez, comenzaron a surgir numerosos artículos, entre ellos un ensayo analítico de Edgar Allan Poe, pretendiendo explicar su funcionamiento. La mayoría de los escritores concluyeron, con razón, que el autómata era un engaño. La caja resultó ser una ilusión óptica que escondía en su interior a un humano que lo manipulaba. Por lo tanto, no se trataba de un autómata real, sino de un humano que jugaba bien al ajedrez.

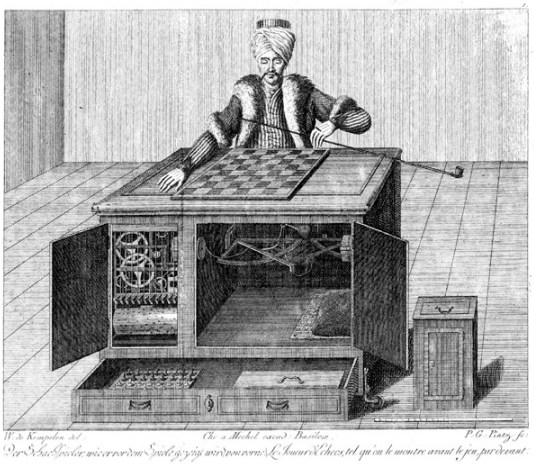

Figura 3.1: El Turco

En **1912** Leonardo Torres y Quevedo construyó un autómata real, llamado "El Ajedrecista" [13]. El mismo utilizaba electroimanes bajo el tablero que le permitían jugar automáticamente un final de rey y torre contra el rey de un oponente humano. Sus movimientos no eran muy precisos y no siempre llegaba al mate en el número mínimo de movimientos, pero sí lograba la victoria en todas las ocasiones. Si bien parece simple el basarse en un conjunto explícito de reglas para realizar movimientos satisfactorios en el final de juego, la idea era muy avanzada para la época. A diferencia de "El Turco", que era un falso autómata operado por humanos, "El Ajedrecista" era capaz de jugar al ajedrez sin intervención humana.

(a) (b)

Figura 3.2: El Ajedrecista

En **1948** Norbert Wiener, matemático estadounidense conocido como el creador de la Cibernética, publicó el libro "Cybernetics or Control and Communication in the Animal and the Machine" [14], dónde describe cómo desarrollar un programa de ajedrez utilizando una búsqueda MiniMax limitada por profundidad y una función de evaluación (ver sección 4.4.3 pág 51). El diario New York Times escribió un artículo sobre Wiener, donde aseguraba que se podría construir un jugador de ajedrez mecánico tan bueno como un ser humano, basándose en la idea de que cada cerebro mecánico provee el conocimiento para construir un mejor cerebro y eventualmente se podrían construir máquinas que superan al mejor cerebro humano en cuanto a capacidad de razonamiento. Según Wiener, los cerebros mecánicos no sólo harían los trabajos por los humanos, sino que también podría resolver problemas tales como controlar bombas atómicas y reconciliar países en conflicto. La única tarea que quedaría para los humanos sería encontrar la forma de detener la máquina en caso que lo quiera destruir.

En **1950** Claude Shannon, ingeniero electrónico y matemático estadounidense, publicó "Programming a Computer for Playing Chess" [1], uno de los primeros artículos sobre el problema de la computadora de ajedrez antes de la existencia de una computadora capaz de jugar al ajedrez. En el artículo, Shannon describe cómo construir una rutina de cálculo adecuada o "programa" capaz de jugar una buena partida de ajedrez utilizando las computadoras de propósito general de la época, convirtiéndose en un visio-

nario y en la base de posteriores desarrollos. Shannon sostenía que si bien el problema del motor de ajedrez no tenía importancia práctica, sí había un gran interés teórico dado que la solución encontrada para este problema sería el inicio del ataque a otros problemas de naturaleza similar pero de mayor importancia. Shannon eligió el ajedrez como problema inicial a resolver dado que:

- el problema esta claramente definido tanto en las operaciones permitidas (los movimientos) como en el objetivo final (jaque mate)
- el problema no es ni muy simple como para ser trivial ni demasiado difícil como para no obtener una solución satisfactoria
- se considera que el ajedrez requiere "pensamiento" para obtener un juego habilidoso, lo cual lleva a admitir la posibilidad de un pensamiento mecanizado, o bien, a restringir el concepto de "pensamiento"
- la estructura discreta del ajedrez se ajusta bien a la naturaleza digital de las computadoras de la época

Según Shannon, una posición de ajedrez debe ser definida incluyendo la siguiente información:

- posición de todas las piezas en el tablero
- a quién le toca mover, Blancas o Negras
- si se han movido el rey y las torres. Esto es importante, ya que por ejemplo al mover una torre, se pierde el derecho al enroque
- registrar el último movimiento, para determinar si es posible realizar una captura al paso legal
- registrar el número de movimientos realizados desde que se capturó o movió el último peón, muy importante para la regla de empate por 50 movimientos realizados sin mover o tomar un peón.

Además, como en cada turno ambos jugadores tienen toda la información de los movimientos anteriores y la única ventaja es haber movido la primera pieza (al contrario que con los juegos de cartas, donde no se pueden ver el material del oponente), cualquier posición del tablero puede ser:

- posición ganadora para las piezas Blancas, es decir, las piezas Blancas pueden forzar la victoria sin importar los movimientos de las piezas Negras
- empate, las piezas Blancas pueden forzar como mucho un empate, sin importar los movimientos de las piezas Negras y viceversa
- posición ganadora para las piezas Negras, es decir, las piezas Negras pueden forzar la victoria sin importar los movimientos de las piezas Blancas.

No es fácil encontrar métodos prácticos para determinar a cuál de las tres categorías anteriores pertenece una posición. En algunos juegos existe una función de evaluación simple $f(P)$ que puede ser aplicada a una posición P y cuyo valor determina a qué categoría pertenece la posición P (ganar, perder o empate). Shannon destaca que si se puede encontrar una función de evaluación $f(P)$ para un juego, entonces es fácil diseñar una máquina capaz de jugarlo exitosamente. Esa función podría definirse como:

$f(P) = 1$ para una posición ganadora,
$f(P) = 0$ para una posición de empate,
$f(P) = -1$ para una posición en la que se pierde la partida.

La función $f(p)$ sería calculada en cada turno que le corresponde mover a la máquina, para cada posible posición resultante de ejecutar cada uno de los movimientos válidos. En base a dicha función, Shannon propone maximizar su valor cuando se evalúa el jugador de turno y minimizarlo cuando se evalúa los posibles tableros para el oponente (como veremos en la sección 4.4.3 pág 51).

Por otro lado, Shannon predijo acertadamente los dos posibles tipos de búsqueda utilizados por cualquier programa, a las que nombró de "Tipo A", y de "Tipo B". Los programas "Tipo A", utilizan una búsqueda basada en la "fuerza bruta", examinando todas las posibles posiciones de cada rama del árbol de movimientos usando el algoritmo MiniMax (ver sección 4.4.3 pág 51). Shannon creía que esto sería muy poco práctico por dos razones: Primero, con aproximadamente 30 movimientos posibles en una posición típica de medio juego, predijo que buscando las 30^6 (más de 700.000.000) posiciones contenidas en los primeros tres movimientos (de ambos bandos, lo que son 6

plies), tardaría aproximadamente 16 minutos, incluso en el caso "muy optimista" en el que el programa evaluara un millón de posiciones por segundo. Después de esta conjetura, se tardó alrededor de 40 años para conseguir esa velocidad. Segundo, se ignoraba el problema de la latencia, ya que el programa trata de evaluar la posición resultante después de todo el intercambio de piezas ocurrido durante todos esos movimientos al final de cada rama del árbol. Los programas de "Tipo A"funcionan así, pero su inconveniente es el gran incremento del número de posiciones necesarias para el análisis, y de este modo el programa se hace cada vez más lento.

Por su parte, los programas de "Tipo B" utilizan una especie de "inteligencia artificial estratégica" tal que en lugar de gastar esfuerzos de procesamiento examinando movimientos malos o triviales, se analizarían únicamente las mejores jugadas de cada posición. Es una estrategia similar a la utilizada por los jugadores humanos, permitiendo al programa analizar los caminos más significativos con mayor profundidad y en un tiempo razonable.

En **1952**, Alan Turing [2], matemático, lógico, científico de la computación, criptógrafo y filósofo británico desarrolla en papel el primer programa de ajedrez. Fue uno de los principales pioneros durante los primeros años de la computación y sostenía que los juegos constituían un modelo ideal para el estudio de la inteligencia en máquinas y como éstas debían construirse. Esta idea es seguida hasta el día de hoy por quienes trabajan en Inteligencia Artificial, pero en los tiempos de Turing la opinión era mirada con bastante escepticismo. En su diseño utilizó una función de evaluación bastante simple en la cual el material era el factor dominante. En caso de ser las evaluaciones de material iguales en los movimientos analizados, los factores posicionales decidían el mejor. Para los factores posicionales se consideraba: Movilidad, Piezas protegidas, Movilidad del Rey, Protección del Rey, Enroque, Posición de peones y Amenazas de jaque o jaque mate. Dado que no tenía a disposición una computadora lo suficientemente potente que pudiera ejecutar su algoritmo, él mismo simulaba a mano el funcionamiento de la computadora, tardando más de hora y media en efectuar un movimiento. Llegó a registrarse una partida en la que el programa (ejecutado a mano) perdió frente a un amigo de Turing [15].

Tanto Turing como Shannon propusieron, con un enfoque mucho más teórico que práctico, las primeras técnicas e ideas básicas para la construcción de máquinas capaces de jugar al ajedrez. Muchas de las ideas propuestas por estos investigadores (búsqueda en profundidad, funciones de evaluación y técnica de juegos) corresponden a la base utilizada actualmente para la realización de software capaz de jugar al ajedrez, lo cual les da un valor histórico muy alto.

En **1964** Richard Ernest Bellman [16], matemático estadounidense e inventor de la programación dinámica, publica el libro *"On the Application of Dynamic Programming to the Determination of Optimal Play in Chess and Checkers"* [17]. En el libro describe los métodos básicos para utilizar la teoría de la programación dinámica con las computadoras de la época, para determinar partidas óptimas en la mayoría de los finales de ajedrez de Peón-Rey y durante todo el juego de Damas. La programación dinámica define teorías matemáticas dedicadas al estudio de procesos de múltiples etapas. Estos procesos se componen de secuencias de operaciones donde el resultado de las etapas precedentes se puede utilizar para guiar el curso de las etapas futuras. Como en ajedrez existen a menudo más de una forma de obtener una misma distribución de piezas en el tablero mediante diferentes secuencias de movimientos (por ejemplo la posición resultante de la secuencia 1.e4 e5 2.f4 d6 es la misma que en la secuencia 1.f4 d6 2.e4 e5), los motores de ajedrez pueden utilizar una técnica derivada de la programación dinámica llamada *Tablas de transposición* (ver sección 6.1), donde se almacenan los resultados de las búsquedas realizadas anteriormente, reduciendo en gran medida el espacio de búsqueda en el árbol de búsqueda (en el ejemplo anterior, el árbol sólo mantendría una de las dos secuencias posibles).

En **1996** IBM presentó una supercomputadora conocida como Deep Blue [18]con la cual intentó ganar al campeón mundial de ajedrez Garry Kasparov en condiciones de torneo. Si bien para ese entonces los programas de ajedrez más avanzados eran capaces de derrotar incluso a los mejores jugadores del mundo en partidas Blitz[1], todavía no se había podido demostrar que fueran capaces de derrotar a los grandes maestros en condiciones de torneo. El propósito de Deep Blue era simple, derrotar al ser humano que ostentara el título de campeón del mundo en una partida a jugarse en condiciones lo más cercanas posibles a las de una partida competitiva o torneo. Deep Blue era una supercomputadora clasificada entre las 300 más poderosas para la época, capaz de evaluar 200 millones de posiciones por segundo. En el torneo, Deep Blue ganó la primera partida en 37 movimientos. Kasparov se recuperó rápidamente y ganó la segunda partida. Las siguientes dos partidas terminaron en tablas. En la quinta partida, Kasparov ofreció tablas en el movimiento 23, pero no fue aceptada y Deep Blue terminó perdiendo la partida. La última partida también fue para Kasparov, terminando el torneo con puntuación 4-2 a favor de éste último.

Figura 3.3: Supercomputadora Deep Blue de IBM

En **1997** se jugó el segundo encuentro, luego que IBM invirtiera un año mejorando a Deep Blue después de su derrota. Este segundo duelo comenzó con un mal resultado para la máquina, perdiendo la primer partida. La segunda partida, terminó en derrota para Kasparov, generando gran controversia ya que Kasparov tuvo un error táctico y asumió que la computadora se percataría de tal hecho. La tercer, cuarta y quinta partida terminaron en

[1]Partida Blitz o relámpago: cada jugador dispone de un máximo de 15 minutos para toda la partida, o bien, todas aquellas partidas con incremento de tiempo por jugada en las que de la suma del incremento multiplicado por 60 y el tiempo inicial de reflexión no se obtenga una cantidad mayor de los 15 minutos. Ver capítulo 4.1.2 pág. 25

empate. La sexta partida generó también grandes controversias dado que Kasparov cometió errores en la defensa Caro-Kann[2], lo que permitió jugar a Deep Blue un sacrificio de caballo arruinando la posición de las negras y Kasparov se rindió en 19 movimientos. Deep Blue fue desarmada poco tiempo después sin que IBM diera una revancha.

[2]Caro-Kann: es una defensa común contra la apertura con el peón de Rey, caracterizada por los movimientos: 1. e4 c6

3.2. Diseño de motores

Desde el punto de vista informático, un motor de ajedrez es un programa que analiza posiciones de ajedrez y toma decisiones para realizar las mejores jugadas. Los motores deciden qué movimientos realizar pero la mayoría no cuentan con una interfaz gráfica de usuario (GUI) propia que les permita interactuar directamente con el usuario. Son más bien aplicaciones de consola que implementan protocolos de comunicación estándar que les permite ser utilizados por diferentes interfaces gráficas como XBoard, WinBoard [4], etc. Esto le permite al usuario jugar contra varios motores sin tener la necesidad de aprender a utilizar una nueva interfaz de usuario para cada uno, ya que utilizarían por ejemplo Winboard para ambos motores. También permite realizar torneos entre diferentes motores. La interfaz gráfica además provee opciones tales como guardar las jugadas de la partida en curso en formato PGN[3](ver sección 4.1.5), retomar partidas salvadas con anterioridad e incorporar libros de apertura.

Los *libros de apertura* contienen una base de datos con movimientos de aperturas de ajedrez que han sido compilados a partir de jugadas realizadas por grandes maestros. Mientras que los movimientos realizados en la partida figuren en el libro de apertura, el motor puede utilizarlo para seleccionar los siguientes movimientos a realizar, lo cual le brinda un camino a transitar que ya ha sido estudiado y ha resultado satisfactorio. En consecuencia, usualmente para los inicios más comunes, los libros de apertura tiene más profundidad de juego que para los inicios no tan usados. Una vez que el motor no encuentra la jugada en el libro de aperturas, comienza a ejecutar sus rutinas de evaluación y búsqueda. Por lo tanto los libros de apertura permiten que el motor ahorre tiempo, gane mayor calidad de juego y proporcione variedad en los inicios de la partida dado que los movimientos son elegidos dentro de las jugadas del libro en vez de ser calculados por él mismo. Esto elimina la necesidad de calcular las mejores jugadas aproximadamente en los primeros 10 movimientos de la partida, donde las posiciones no son muy determinantes y computacionalmente costosas de evaluar, permitiendo que el motor juegue mejor requiriendo menos recursos que si tuviera que calcular él mismo los movimientos.

[3]Portable Game Notation, texto sin formato utilizado por programas de computadora para grabar los movimientos (en notación algebraica) y otros datos, como contrincantes y resultados, de partidas de ajedrez.

Como ya mencionamos anteriormente, el diseño de motores de ajedrez implica dos componentes independientes entre sí. Uno es el motor propiamente dicho, que se encarga de evaluar cuál es la mejor jugada a realizar dentro del conjunto de jugadas posibles siguiendo alguna lógica de evaluación y comportamiento. El segundo componente es la implementación de algún protocolo estándar que permita utilizar interfaces gráficas conocidas (componente encargado de interactuar con el jugador humano o de realizar torneos entre diferentes motores). En la construcción de motores, se deben diferenciar las tres etapas principales de una partida (ver sección 4.1): apertura, medio juego y final; y el motor debe comportarse de forma acorde en cada una de ellas.

Para la etapa inicial de la partida, se pueden utilizar los ya mencionados libros de apertura. Como veremos más adelante en el captítulo 5 (página 70), las posiciones iniciales generan dificultades al momento de ser analizadas por un motor debido a la asimetría que se da en la valoración del tablero cuando se analiza desde el punto de vista de blancas o negras (generalmente es más favorable para blancas).

Es en el transcurso del medio juego donde el motor deberá mostrar su destreza para poder controlar y dominar a su rival. En esta etapa se utilizarán todas las algoritmias de las cuales disponga nuestro motor para seleccionar, de todas las jugadas posibles, la que le de mayor rédito a futuro. Este es el punto de mayor exigencia tanto para el motor de ajedrez, como para el hardware sobre el que está siendo ejecutado, llegando en algunas ocasiones a provocar recalentamientos que conlleven a un apagado involuntario.

En la última etapa del juego, cuando ya está en claro que un jugador posee una posición dominante y/o se tiene un recuento de piezas bajo, entran a jugar las técnicas de finales de partida. Una de las técnicas de finales de partida puede ser utilizar una base de datos del estilo de los libros de apertura, en los cuales ya se tiene una valoración para cada movimiento y el motor solamente debe escoger el que más le convenga. Otra técnica puede ser la utilización de una algoritmia particular (enfocada en finales de juego) que analiza las posibles jugadas desde una óptica más agresiva, que en la mitad del juego, en busca del preciado “jaque mate”.

Como en la mayoría de los deportes y juegos que practica cualquier ser

humano, existe siempre cierto grado de competencia o búsqueda de ranking entre diferentes contendientes; los motores de ajedrez poseen sus propias competencias y rankings. Uno de los rankings más conocidos y fiables es la CCRL (Computer Chess Rating List) [19], en el cual se listan los más de trescientos motores de ajedrez ordenados por puntajes según diferentes tipos de partidas donde por ejemplo 40/40, correspondiente a cuarenta movimientos en cuarenta minutos, 40/4 corresponde a cuarenta movimientos en cuatro minutos y 404FRC, que es equivalente a 40/4 pero con aperturas aleatorias y cambio de bandos. Es importante destacar que todos los motores evaluados son ejecutados en una máquina particular (actualmente un AMD Athlon X2 64 4600+ (2.4Ghz)), con un libro de aperturas general e imposibilitando a los motores realizar algún tipo de procesamiento mientras el motor oponente está "pensando" su jugada.

3.3. Motores Actuales

Hoy en día, gracias a los avances en las técnicas de programación y mejoras en el hardware de los equipos computacionales, los motores de ajedrez se han vuelto accesibles al consumidor promedio. Existen motores de ajedrez disponibles para descargar desde Internet, algunos previo pago y otros de forma gratuita. Dentro de los motores de libre acceso se destacan Crafty, GNU Chess, StockFish, Fritz y Houdini.

Cualquiera de estos programas tienen la posibilidad de derrotar y sobrepasar, en cuanto al nivel de juego, a cualquier jugador humano cuando son ejecutados en un computador personal promedio.

Cada año que pasa, los motores de ajedrez incrementan su capacidad de juego de forma significativa, impulsados por dos aspectos fundamentales. El primero está dado por el incremento en la velocidad de procesamiento del hardware permitiendo llegar a mayor profundidad en el análisis de las jugadas para un determinado tiempo de procesamiento. El segundo aspecto está dado por las mejoras en las técnicas de programación, que permiten a estos motores tomar mejores decisiones sobre qué líneas de juego analizar para tener un mejor entendimiento de la posición y en base a estos datos, decidir qué movimiento elegir.

Capítulo 4

Desarrollo del proyecto

El objetivo principal de nuestro proyecto fue desarrollar y optimizar un motor de ajedrez, partiendo de un motor base a elección. En consecuencia, se realizó un relevamiento de la literatura existente relativa al desarrollo de motores de ajedrez. En base a la información encontrada, se desarrolló y optimizó un motor de ajedrez al cual nombramos "***Joli***", tomando como base el motor "***Winglet***" [3] elegido por nosotros, con el compromiso de mantener un juego sólido y de fácil extensibilidad. Luego se midió el potencial de juego del programa desarrollado, "*Joli*", contra el motor base *"Winglet"*, así como con voluntarios humanos. El proyecto fue orientado por el Dr. Ing. Pablo Romero (Gr. 3, Doctor en Informática) y el Ingeniero Gonzalo Varalla quien posee amplio conocimiento en motores de ajedrez, y aportó sugerencias para medir la solidez del motor desarrollado.

A continuación, en las siguientes secciones, detallaremos cada una de las etapas transitadas para llegar a la presentación de ***"Joli"***, el primer motor de ajedrez implementado en la Facultad de Ingeniería.

4.1. Relevamiento bibliográfico

Para encaminarnos en la búsqueda de un buen motor de ajedrez a utilizar como base de nuestro desarrollo, se realizó un relevamiento de la literatura relativa a la creación y optimización de motores de ajedrez. Encontramos mucha información, pero uno de los artículos más destacados fue *"Programming a Computer for Playing Chess"* de Claude E. Shannon (ver sección 3.1, historia de motores), el cual fue determinante y nos proveyó gran parte de los conocimientos que necesitábamos sobre la creación de motores de ajedrez.

Por otro lado, en cuanto a la selección del motor base, entre una gran variedad de material encontrado, nos entusiasmó mucho uno en particular escrito por Stef Luijten, llamado *"Writing a chess program in 99 steps"* [3], el cual brinda una guía paso a paso para que los entusiastas de la programación de ajedrez, como nosotros, puedan implementar un motor de ajedrez. Stef Luijten escribió la guia paso a paso en base a artículos de Robert Hyatt, autor del conocido motor de ajedrez Crafty, y de Bruce Moreland, programador de motores de ajedrez y quien publicó los fuentes de su motor Gerbil para fines educativos. El motor resultante es una versión "libre" bajo GNU (General Public License) del motor "*Wing*", propiedad de Stef Luijten quien lo nombró "*Winglet*".

En cuanto a la mejora de la función de búsqueda otro artículo que nos ayudó fue *"Evolution of an Efficient Search Algorithm for the Mate-In-N Problem in Chess"* de Ami Hauptman y Moshe Sipper [20], dónde se explica cómo desarrollar un algoritmo de búsqueda eficiente utilizando programación genética, evolucionando la función de búsqueda en el árbol de búsqueda para resolver el problema de *Mate-In-N*, el cual corrobora si existe un movimiento tal que para cualquier respuesta del oponente se pueda dar mate en N jugadas futuras.

Luego de leer varios artículos creemos que es necesario definir algunas nomenclaturas utilizadas en la implementación de motores de ajedrez que pueden ayudar a comprender mejor las siguientes secciones:

4.1.1. Etapas del juego

Las partidas de ajedrez se pueden dividir en 3 etapas. "*Apertura*" equivalente a las primeras jugadas, donde las piezas van saliendo de sus casillas iniciales; "*Medio juego*", donde los dos bandos aún tienen muchas piezas y peones, y éstos entran en intenso conflicto. Y "*Final*" donde quedan pocas piezas y peones y se cambia la estrategia para buscar con más énfasis el jaque mate.

4.1.2. Ritmos de juego

La duración de cada partida puede ser:

- *Partida Blitz o relámpago*: cada jugador dispone de un máximo de 15 minutos para toda la partida, o bien, todas aquellas partidas con incremento de tiempo por jugada en las que de la suma del incremento multiplicado por 60 y el tiempo inicial de reflexión no se obtenga una cantidad mayor de los 15 minutos.

- *Partida rápida*: aquella en la cual el tiempo para cada jugador está entre 15 y 60 minutos, o si se juega con incremento, aquella cuyo tiempo de reflexión inicial, más la suma del incremento multiplicado por sesenta, quede comprendida en ese intervalo. Normalmente este ritmo de juego se utiliza en desempates de torneos jugados con ritmo normal, y su tiempo se fija en 25 minutos.

- *Partida estándar (ritmo clásico)*: el tiempo de reflexión por jugador es mayor de 60 minutos. Este es el ritmo de juego más usado a nivel magistral. En torneos con gran cantidad de participantes, suele existir un control de 90 minutos por jugador más 15 segundos de incremento por jugada, mientras que en torneos de élite se asignan controles de tiempo para cierto número de jugadas, más un tiempo para el final de la partida. También se las denomina partidas de ritmo normal o lento.

- *Sin reloj*: cada jugador puede disponer de todo el tiempo que desee para pensar sus jugadas.

4.1.3. Notación Algebraica de Partidas de Ajedrez [8]

La Notación Algebraica es una representación escrita de los movimientos que se van sucediendo a lo largo de una partida de Ajedrez, y desde 1997 es el único sistema de notación oficial, reemplazando al sistema de notación Descriptiva. La identificación de los escaques del tablero se realiza de forma única mediante dos caracteres: una letra seguida de un número. El primer caracter identifica la columna del tablero y el segundo caracter identifica la fila. Para identificar una columna, se utilizan las letras de la **a** a la **h** de forma correlativa a partir del margen izquierdo del tablero visto desde el lado del jugador en blancas (en minúscula). La numeración de las filas es de forma análoga y parte desde la misma casilla, en dirección a las piezas negras etiquetando con los números del **1** al **8** como se muestra en la figura 4.1.

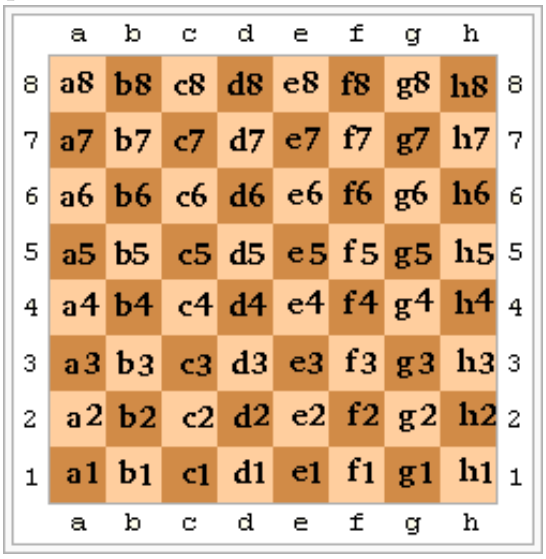

Figura 4.1: Identificación de escaques

Las piezas se identifican con una letra, en mayúscula y dependiendo del idioma en el que se esté anotando, pueden ser, por ejemplo:

- Español: R = Rey, D = Dama, T = Torre, A = Alfil, C = Caballo
- Inglés: K = King, Q = Queen, R = Rook, B = Bishop, N = Knight

Los peones no tienen asociada ninguna letra y solamente se utiliza la información relativa al movimiento para identificarlos.

El movimiento básico es descrito mediante la letra de la pieza que se mueve (recordar que para peones no se utiliza letra alguna) y la casilla de destino. Un movimiento común al inicio de una partida puede ser "Kc3", lo que significa que estamos avanzando el caballo blanco desde la casilla b1 hasta la casilla c3. Cuando se presenta el caso de movimientos ambiguos donde se tiene más de una pieza del mismo tipo que pueda acceder a determinada casilla, si solamente utilizamos la casilla de destino como es usual, no tendríamos forma de

identificar fehacientemente cuál de las piezas realizó dicho movimiento. En ese caso estamos ante un movimiento ambiguo. Imaginemos por ejemplo dos peones blancos, uno en **a4** y otro en **c4**, y un peón negro en **b5**. Si realizamos el movimiento de captura con el peón de **a4** a **b5**, tendríamos entonces el movimiento descrito por **xb5** (la letra x indica una captura), el problema en este punto es que si tratamos de reproducir los movimientos a partir del caso descrito, no sabríamos cuál de los dos peones realizó la captura solamente con la información relativa al destino de la pieza. Para solucionar este problema, se definieron reglas para resolver ambigüedades:

- Si las piezas pueden ser diferenciadas por la columna en la que están posicionadas al momento de partida del movimiento, luego de la letra de la pieza, se inserta la letra correspondiente a dicha columna.
- Si la ambiguedad continúa, se intenta identificarlas por la fila que ocupan, realizando el mismo procedimiento al momento de anotar el movimiento, se inserta el número de columna correspondiente inmediatamente después de la letra de la pieza que realiza el movimiento.
- Si la ambiguedad no se resolvió, se agregan ambas coordenadas a continuación de la letra de la pieza, describiendo completamente las filas y columnas origen y destino de la pieza.

Volviendo al ejemplo de los peones, tendríamos que el movimiento sin ambigüedad sería **axb5**, indicando la columna origen mediante la letra **a**, la captura realizada con la letra **x** y finalmente la casilla destino **b5**.

Para detallar los movimientos especiales, se tienen diferentes secuencias de caracteres que facilitan su identificación. Los enroques se detallan O-O y O-O-O respectivamente para el enroque corto (del lado del Rey) o el enroque largo (para el lado de la Dama). Como se mencionó antes, las capturas se anotan con la letra x y las promociones de peones a piezas se denotan en base a la casilla en la que fue promovido, un signo "=" y la pieza en la que se convirtió. Por ejemplo, al promover un peón al llegar a g8, tendremos g8=Q. Para denotar los eventos de jaque se utiliza el signo "+" y para el jaque mate, pueden utilizarse "++" o "#" de forma indistinta.

Por último, la notación de comentarios es variada y a continuación se pueden ver algunos de los símbolos utilizados:

- !: jugada buena
- ?: jugada mala
- !!: jugada muy buena
- ??: jugada muy mala
- !?: jugada interesante
- ?!: jugada dudosa
- +/=: ligera ventaja blanca
- =/+: ligera ventaja negra
- ± (ó +/-): ventaja blanca
- -/+: ventaja negra
- +-: ventaja decisiva blanca
- -+: ventaja decisiva negra
- N: novedad teórica
- {texto} comentarios de texto en el documento se escriben encerrados entre llaves "{" y "}"

4.1.4. FEN (Forsyth–Edwards Notation) [21]

Notación estándar para describir una disposición de piezas particular sobre tablero. Su propósito es proveer toda la información necesaria para reiniciar una partida a partir de una posición determinada del tablero. Es un archivo de extensión .fen y esta compuesto por una línea de texto que contiene seis campos separados por un espacio en blanco. Los campos son:

1. Posición de las piezas (desde la perspectiva de las Blancas). Se describe cada fila comenzando por la fila ocho y terminando en la primera. Dentro de cada fila se describe el contenido de cada escaque desde el escaque "a" hasta el "h". Las piezas dentro de los escaques de una fila son representados siguiendo la notación algebraica. Por lo tanto,

cada pieza es identificada por una sola letra a partir de su nombres en Inglés (Caballo = “N”, Alfil = “B”, Torre = “R”, Dama = “Q” y Rey = “K”). Las piezas blancas se representan con letras mayúsculas (“PNBRQK”), mientras que las negras, en minúsculas (“pnbrqk”). Las casillas en blanco se anotan usando los dígitos del 1 al 8 (número de casillas en blanco), y el carácter “/” sirve para separar las 8 filas.

2. Color al que le toca mover. “w” significa que mueven las blancas y “b” las negras

3. Posibilidad de enroque. Si el enroque ya no es posible para ninguna de las partes, se coloca el carácter “-”. De lo contrario, figuran las letras “K”, “Q”, “k” y/o “q”, tales que: “K” indica que es posible el enroque del lado del rey blanco), “Q” que es posible el enroque del lado de la dama blanca, “k” que es posible el enroque del lado del rey negro, y “q” indicando que es posible el enroque del lado de la dama negra.

4. Posibilidad de captura al paso de un peón en notación algebraica. Si no hay posibilidad de captura al paso de ningún peón, se colocará el carácter “-”. Si un peón acaba de hacer una jugada de 2 casillas, se pondrá la casilla que está “detrás” del peón. Ésta se deberá poner independientemente de si hay o no algún peón en condiciones de realizar una captura al paso.

5. Plys desde la última captura o avance de peón. Esto se utiliza para determinar si se termina el partido por tablas en virtud de la regla de los cincuenta movimientos.

6. Número total de movimientos de la partida. Tiene el valor de 1 para la posición inicial y se va incrementando con cada movimiento de las negras.

4.1.5. PGN (Portable Game Notation) [22] [23]

Es un archivo con extensión .pgn que contiene texto sin formato utilizado por programas de computadora para grabar los movimientos (en notación algebraica) y otros datos de partidas de ajedrez como contrincantes, resultados, fecha, etc. La estructura de un PGN está pensada para facilitar la escritura y lectura de los usuarios humanos y para facilitar el parseo y generación de los programas de computadora. Existen dos formatos de PGN, formato de importación escrito a mano y por lo tanto muy variado; y el formato de exportación que es el generado por los programas, el cual es más estricto en su estructura.

La estructura del PGN comienza con un conjunto de pares de etiquetas del estilo [nombre-etiqueta "valor"], seguido por la lista de movimientos en notación algebraica (y opcionalmente comentarios en cada jugada). Para el almacenamiento de archivos PGN, es obligatorio contar con al menos 7 etiquetas llamadas STR (Seven Tag Roster), las cuales deben figurar antes que cualquier otra etiqueta y en el siguiente orden:

1. Evento: Nombre del torneo o de la partida
2. Sitio: ubicación del evento en el formato "Ciudad, Región, PAÍS", donde PAÍS debe ser el código de tres letras correspondiente al país según el Comité Olímpico Internacional. Por ejemplo "Montevideo, Montevideo, URU"
3. Fecha: fecha de inicio del juego, en formato YYYY.MM.DD (se utiliza "??" si no se conocen los datos).
4. Ronda: número de ronda de la partida dentro del evento.
5. Blancas: el jugador de las piezas Blancas, en formato "Apellido, Nombre".
6. Negras: el jugador de las piezas Blancas, en formato "Apellido, Nombre".
7. Resultado: resultado del partido, que sólo puede tener 4 posibles valores: "1-0" (ganan las Blancas), "0-1" (ganan las Negras), "1/2-1/2" (empate), or "*" (otros, por ejemplo la partida no ha terminado).

Otras etiquetas opcionales pueden ser: “Time”, hora de inicio del partido en formato “HH:MM:SS” y “FEN” donde se indica la posición inicial del tablero en notación FEN permitiendo continuar partidas no finalizadas.

4.1.6. Bitboard

Un bitboard es una estructura de datos utilizada comúnmente en sistemas informáticos que implementan juegos con tableros, como el caso de los motores de Ajedrez. Los bitboards son esencialmente un conjunto finito de hasta 64 elementos (p.ej.: una palabra de 64 bits que representa todos los escaques de un tablero de Ajedrez) utilizándose un único bit para representar a cada uno de los escaques. Este tipo de representación presenta significativas ventajas sobre otros tipos de representaciones dado que permiten operaciones a nivel de bits, las cuales pueden ser ejecutadas en mucho menos tiempo que las operaciones normales. Otra importante ventaja es la eficiencia en cuanto al espacio de memoria requerido para su almacenamiento.

4.1.7. Protocolos de comunicación [4]

Los protocolos de comunicación permiten que los motores de ajedrez funcionen con una interfaz gráfica que ha sido creada por otra persona. Para ello tanto el motor como la interfaz deben comunicarse de alguna forma y con determinadas reglas, es decir que debe existir un protocolo de comunicación entre la interfaz y el motor.

Chess Engine Communication Protocol [4] es el primer protocolo de comunicación abierto diseñado por Timothy Mann [5], el cual permite comunicar un motor de ajedrez con su interfaz de usuario llamada ***XBoard*** (para sistemas operativos Unix\Linux). Inicialmente fue diseñado para comunicarse sólo con *GNU Chess*, uno de los motores de ajedrez más antiguos y gratuito, el cual no cuenta con una interfaz de usuario. Utiliza la consola para desplegar el tablero mediante letras y los movimientos deben ser ingresados a través del teclado. Como *GNU Chess* sólo acepta texto de entrada y texto de salida, la primera versión del protocolo no era más que el comportamiento de la interfaz de línea de comandos de *GNU Chess*. Por lo tanto la interfaz le comunicaba al motor, mediante líneas de texto, los movimientos y comandos necesarios y por otro lado parseaba el texto de salida para representarlo en un tablero gráfico de ajedrez. Dado el éxito de la interfaz *XBoard*

que permitía brindar un ambiente agradable para el jugador, Timothy Mann aprovechó dicha interfaz y estableció un protocolo de comunicación entre su interfaz y otros posibles motores, al cual llamó protocolo de comunicación *XBoard*. Con este nuevo protocolo se cambió la forma de crear los programas de ajedrez al dividirse en parte gráfica (GUI) y parte inteligente que sabe jugar ajedrez (motor). Luego se implementó la versión para windows llamada ***WinBoard***, pasándose a llamar protocolo *Winboard* hasta la actualidad. El segundo motor que adoptó el protocolo, en el año 1995 y posiblemente el más conocido motor compatible con *XBoard*, es el programa *Crafty*, del profesor Bob Hyaat. A partir de ahí han ido surgiendo una multitud de motores que utilizan el protocolo *XBoard*.

Más recientemente, por el año 2000, ha surgido un segundo protocolo de comunicación llamado ***Universal Chess Interface (UCI)*** [24], siendo su mayor impulsor el programador del conocido programa Shredder [25]. Programa comercial que ganó hace unos años muchos campeonatos mundiales de motores. El protocolo de comunicaciones *UCI* también es de acceso libre y puede verse como un rival del protocolo *XBoard/WinBoard*. Intenta corregir y ampliar alguno de los problemas que tiene el protocolo *XBoard*, por ejemplo asigna tareas a la interfaz que tradicionalmente eran manejadas por el motor. En particular, en *UCI*, se espera que el libro de aperturas sea manejado por la interfaz, seleccionando movimientos del libro hasta que se juegue un movimiento que se encuentra fuera del libro, y recién en ese momento se invoca al motor para el cálculo de la posición resultante. Pocas interfaces y motores soportaban este nuevo protocolo hasta que la compañía de software de ajedrez, Chessbase, comenzó a soportar *UCI* en el año 2002. En el año 2007 ya existían más de 100 motores que soportaban *UCI*.

En la actualidad muchos motores soportan ambos protocolos, y cada protocolo tiene su soporte.

4.2. Selección del motor base

Luego de relevar información referente a motores de ajedrez, nuestro punto de partida para comenzar a desarrollar nuestro motor, fue el contar con un motor de ajedrez que sea de código abierto, plausible de modificar y que cuente con un conjunto mínimo de funcionalidades básicas para un motor de ajedrez permitiendo minimizar nuestro tiempo de desarrollo, dados los tiempos del proyecto. Logrando así focalizar nuestra atención en la implementación y mejora de las funciones de búsqueda y evaluación. Las funcionalidades básicas que buscábamos en un motor por ejemplo eran: buena definición del tablero y representación de las piezas (eficiente en cantidad de memoria requerida y fácil de manejar). Otra funcionalidad buscada era que implemente algún protocolo de interfaz gráfica para permitir interactuar mediante interfaces utilizadas comúnmente por los jugadores de ajedrez (como Winboard, Arena, etc).

Una gran decisión que tuvimos que tomar, que condicionaría el desarrollo futuro de nuestro motor, fue la selección del lenguaje y plataforma en la que estuviese implementado, ya que esta elección afectaría muchos factores, desde el grado de dificultad al momento de realizar modificaciones en el código, hasta el propio desempeño del motor. Tomando los requerimientos antes mencionados, acotamos nuestra búsqueda a unos pocos motores que cumplieran con esos requisitos y finalmente optamos por seguir la guía paso a paso "*Writing a chess program in 99 steps*" [3], con la cual obtuvimos nuestro motor base nombrado *Winglet* [3] por su creador Stef Luijten. *Winglet* es un motor de ajedrez de código abierto implementado en lenguaje C++ que además de tener implementadas las funcionalidades básicas de un motor de ajedrez, posee una interfaz por línea de comando y puede interactuar con interfaces gráficas que soporten el protocolo *Winboard*. Otro punto a favor es que posee una amplia documentación donde apoyarnos en caso que fuese necesario.

Luego de haber definido el motor "base" y de haber seguido los pasos para la creación de nuestro motor al cual nombramos "***Joli***", implementamos un conjunto de funcionalidades que serían de utilidad al momento de generar nuestras funciones de evaluación y búsqueda, así como también para familiarizarnos tanto con el ambiente de desarrollo como con el código del motor.

En cuanto a las funciones de búsqueda y evaluación de tableros (ver sección 4.4.3), ayudados en gran parte por ambos tutores y por material relevado sobre el tema, fuimos conociendo las diferentes métricas de evaluación, funciones de búsqueda "clásicas" en ajedrez y algo no menos importante, la notación utilizada para registrar las partidas, la cual también servirá luego al momento de probar diferentes funciones de evaluación.

4.3. Detalles del motor implementado

Para la implementación de "***Joli***", utilizamos el entorno de desarrollo ***Microsoft Visual Studio Express 2012*** [26] para C++, ya que cuenta con su propio compilador y otras herramientas como IntelliSense y opción de Debug que facilitan la implementación. Además, como el código fuente del programa está compuesto por muchos archivos, vimos que era necesario utilizar un repositorio tanto para el versionado de dicho código como para permitir la implementación en conjunto por ambos integrantes del proyecto. El repositorio elegido para trabajar fue ***Git*** [27] diseñado por Linus Torvalds (creador del kernel del sistema operativo Linux) y utilizado actualmente por muchos desarrolladores reconocidos. *Git* puede ser integrado al *Visual Studio* facilitando aún más nuestro trabajo en equipo. Se utilizaron las versiones gratuitas tanto del *Visual Studio* como del *Git*, descargándolas directamente de los sitios web oficiales de cada uno.

A continuación explicaremos algunos detalles de implementación referentes a la representación del tablero, las piezas y sus movimientos.

4.3.1. Representación interna del tablero

La representación del tablero se basa en artículos de Robert Hyatt (autor de Crafty) y consiste en utilizar palabras de 64 bits (*Bitboards*, ver sercción 4.1.6), tal que cada una de las palabras almacena todas las posiciones de un tipo de pieza. Puesto que hay 12 piezas diferentes, se necesitan 12 Bitboards para representar cada posición en el tablero de ajedrez. Cada escaque es representado por uno de los 64 bits de la palabra, donde el bit con valor 1 indica la presencia del pieza y con valor 0 indica que hay otro tipo de pieza o el escaque se encuentra vacío. En la figura 4.2 se pueden ver los 12 Bitboards correspondientes a las

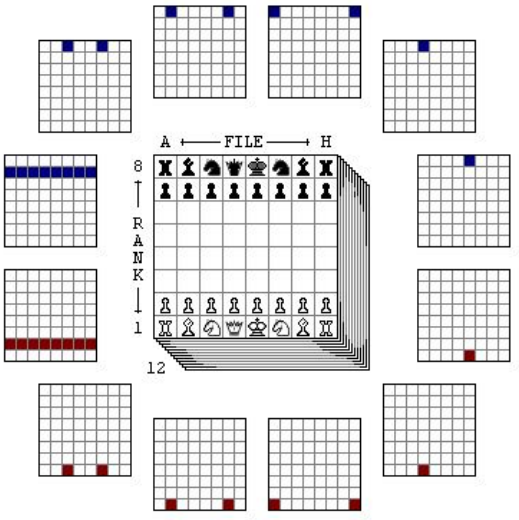

Figura 4.2: Bitboards - tablero

piezas y los peones donde un escaque coloreado corresponde a un bit con valor 1, por lo tanto vemos que si se ubican los Bitboards unos sobre otros se obtiene la representación total del tablero.

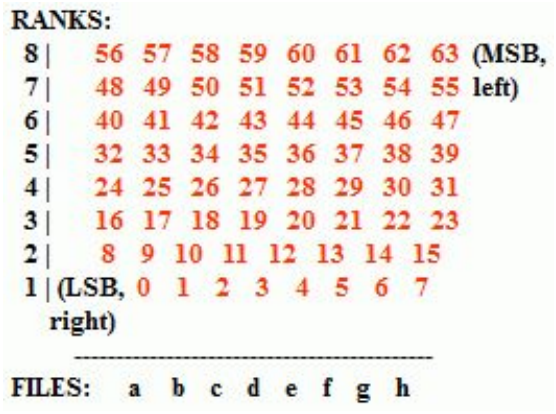

Figura 4.3: Bitboard - numeración

Para saber la correlación entre los bits del Bitboard y los escaques del tablero, es necesario introducir una convención de numeración. Como se puede ver en la figura 4.3, la numeración es la siguiente: el bit #0 es el bit menos significativo (LSB, de más a la derecha) de la palabra de 64 bits y el bit #63 es el bit más significativo (MSB, de más a la izquierda). Además, la correspondencia con los escaque es la siguiente: a1 = bit #0, h1 = bit #7, a8 = bit #56 y h8 = bit #63. Si $FILE$ y $RANK$ se numeran desde 1 a 8, entonces el $ESCAQUE(0..63)$ se puede calcular de con la siguiente fórmula:

$$ESCAQUE = 8 * RANK + FILE - 9$$

Además, se tienen algunas operaciones rápidas sobre los bits que podemos realizar en los Bitboards, como por ejemplo:

- para obtener el bitboard de todas las piezas blancas, podemos utilizar el operador de bits OR (|) tal que:
  $whitePieces = whiteKing \mid whiteQueens \mid whiteRooks \mid whiteBishops \mid whiteKnights \mid whitePawns$

- para obtener todos los escaques ocupados, podemos calcular:
  $escaquesOcupados = whitePieces \mid blackPieces$

En total existen 6 operadores en C++, que son utilizados en combinación con Bitboards y permiten la utilización de máscaras. Los operadores listan a continuación:

- & operador AND, compara dos valores y retorna un valor que tiene

sus bits en 1 si y sólo si, ambos valores que se están comparando tienen esos bits en 1

- | operador OR, compara dos valores y retorna un valor que tiene sus bits en 1 si alguno de los valores comparados tiene esos bits en 1.
- $\wedge$ operador XOR, compara dos valores y retorna un valor que tiene sus bits en 1 si uno y sólo uno de los valores comparados tiene el bit en 1
- $\sim$ operador complemento a 1 o inverso, actúa sólo sobre un valor invirtiendo todos los bits en 0 a bits en 1 y viceversa.
- $>>$ operador de desplazamiento a la derecha, desplaza los bits desde el más significativo (MSB) hacia el menos significativo (LSB), la cantidad de posiciones especificada.
- $<<$ operador de desplazamiento a la izquierda, desplaza los bits desde el menos significativo (LSB) hacia el más significativo (MSB), la cantidad de posiciones especificada.

Los Bitboards ofrecen ventajas en la velocidad de procesamiento contra representaciones matriciales en dos funcionalidades vitales del programa. La primera (y principal) es en la velocidad de generación de movimientos; su rapidez se debe a que no requiere ciclos para generar los movimientos de una pieza ya sea que ésta se mueve por filas, columnas o diagonales (se verá más adelante en la sección 4.3.2). La segunda ventaja se puede ver en la velocidad de la función de evaluación, donde por ejemplo, podemos chequear si el peón en "e4" está adelantado ("passed pawn") con una sola operación, sin necesidad de hacer ciclos sobre los escaques d5, d6, d7, e5, e6, e7, f5, f6 y f7 para corroborar si alguno de estos está ocupado por un peón enemigo. La estructura del tablero contendrá los doce bitboards, junto con información adicional como los enroques disponibles para cada jugador.

Algunas variables útiles serán computadas incrementalmente, lo que significa que su valor se irá modificando en las funciones "makeMove" y "unmakeMove" (hacer y deshacer un movimiento). Tenemos por ejemplo la variable "material" que es donde almacenamos el balance total del material ($peón = 100$, $caballo = 300$, etc usando valores positivos para los blancos y negativos para los negros). Si una pieza es capturada, o un peón es coronado, el valor de la variable material será actualizado de forma acoorde. Esta

información estará disponible de forma directa en la función de evaluación, sin necesidad de realizar cálculos. Esta forma de actualización incremental del balance de material en las funciones par hacer y deshacer movimientos es también ventajosa cuando queremos conocer este valor a medida que descendemos por el árbol de búsqueda, ya que siempre está actualizado y no es necesario hacer una evaluación completa de la posición, economizando tiempo. Otro vector que es conveniente introducir es el vector *square*, que está compuesto por 64 valores de tipo entero e indicará qué pieza ocupa cada uno los casilleros.

Las piezas se identifican con cuatro bits de la siguiente forma: el color está descrito en el primer bit, si la pieza es blanca, será de la forma 0XXX y si la pieza es negra, tendrá la numeración comenzando con 1. Si una pieza puede deslizarse por el tablero (filas, columnas o diagonales), su segundo bit estará en 1, de la forma X1XX, de lo contrario valdrá 0, teniendo la forma X0XX. Si la pieza desliza por filas o columnas, tendrá la forma X11X y si desliza diagonalmente será de la forma X1X1. Los peones tendrán los valores 0001 y 1001 para blancas y negras respectivamente; el valor del rey será 0010 y 1010 en el mismo orden de colores. El Caballo, Alfil, Torre y Reina tendrán los valores 0011, 0101, 0110 y 0111 respectivamente, para blancas; para piezas negras solamente se reemplaza el bit de más a la izquierda con 1.

El valor material de las piezas en el tablero se expresa en centipawns, que es la unidad de medida utilizada en Ajedrez. Un centipawn equivale a 1/100 de peón (100 centipawns equivalen a 1 pawn o peón). Si bien estos valores no juegan ningún papel formal en el juego, son útiles para el jugador común y esenciales en el Ajedrez por computadora para poder evaluar las posiciones. Las piezas generalmente tienen un valor entero en pawns, pero el uso de centipawns permite un manejo más estratégico de las características de una posición. En nuestro motor, a excepción del Alfil, y el Rey todas las demás piezas tienen valores enteros en pawns; el Peón mantiene su valor clásico de 100 centipawns, el Caballo 300 centipawns y el Alfil 350 centipawns. Este valor del Alfil (superior al valor del Caballo) fue elegido para favorecer la preservación de los mismos en el medio juego ya que observamos que nuestro motor era propenso a perderlos de forma poco beneficiosa, influyendo negativamente sobre el final de la partida. El valor de la Torre se configuró a 500 centipawns, el de la Reina en 1000 y por último, el valor del Rey fue definido en 9999.

4.3.2. El generador de movimientos

El principio fundamental de la generación de movimientos en un tablero con representación de Bitboard, es no calcular los movimientos sino buscarlos en una tabla. Hay diferentes métodos, pero todos buscan un balance entre velocidad y memoria utilizada, dependiendo del uso que se le vaya a dar al motor de Ajedrez. Aquellos motores que se implementen para dispositivos móviles tendrán tablas compactas, requiriendo un costo computacional más elevado para calcular la tabla completa donde buscar los movimientos. Otro método es mantener tablas grandes, donde se encuentren todos los posibles movimientos y que requieran un costo computacional mínimo para obtener los movimientos buscados. En nuestro caso particular, hemos optado por mantener las tablas grandes, debido a que no es tan escaso el recurso memoria.

4.3.3. Estructura del movimiento

Un movimiento puede ser representado mediante el formato **from** (desde) y **to** (hasta), con dos enteros positivos en el rango de 0 a 63, consumiendo 6 bits cada uno. Como también se necesita tener la posibilidad de deshacer un movimiento, las piezas capturadas necesitan también estar incluidas en él; el identificador de piezas será un número entre 0 y 15, por lo que precisaremos solamente cuatro bits.

En caso de que estemos ante una promoción de un peón, también necesitamos almacenar la pieza en la que se "transformará" este peón (Torre, Caballo, Alfil o Reina), por lo que necesitaremos agregar 4 bits más. También se agregará un campo para identificar la pieza que estamos moviendo.

Adicionalmente debemos procurar una forma de registrar otros movimientos "especiales" tales como capturas al paso y enroques, para los primeros, utilizaremos los mismos bits que se utilizan para detallar una promoción de peón. Para los enroques, no es necesario denotarlo de alguna forma especial, ya que si el rey se mueve más de un escaque, estamos frente a un enroque. De todas formas, puede utilizarse el espacio destinado a detallar una promoción de peón para este fin.

Generador de movimientos de caballos

La idea básica para un generador de movimientos es que todos los movimientos estén pre-calculados y almacenados en una tabla de búsqueda para que el generador simplemente los utilice sin necesidad de calcularlos.

Primero debemos crear un bitboard objetivo con todos los posibles destinos, excluyendo las casillas ocupadas por piezas de nuestro bando. Como ya sabemos cuál será la pieza que vamos a mover, vamos almacenando este valor en la estructura del movimiento.

Después debemos recorrer cada uno de los caballos de nuestro bando y para cada uno de ellos, aplicamos la función AND bit a bit con el bitboard de ataque para la pieza, utilizando la posición en la que se encuentra. El bitboard de ataque es una tabla de búsqueda precalculada de las 64 tablas posibles que hay para los caballos, una para cada escaque de origen.

Luego de aplicar el operador AND entre el bitboard de ataque y nuestro bitboard generado para el caballo actual, obtenemos todos los posibles escaques destino para una determinada posición de partida. Esta técnica de generación de movimientos es una de las grandes ventajas del uso de bitboards.

Luego con un ciclo, vamos recorriendo los escaques disponibles y generando cada uno de los movimientos ingresando el casillero destino y la pieza capturada, en caso que corresponda. Con esta información del movimiento, procedemos a almacenarlo en el buffer de movimientos e incrementar el largo del mismo, comenzando nuevamente el proceso de búsqueda del siguiente movimiento para esta misma pieza. En caso que no hayan más movimientos posibles, pasamos a la pieza siguiente (en este caso, el siguiente caballo).

Generador de movimiento del Rey

El generador de movimientos para el rey es similar al del Caballo. Primero comienza generando un bitboard destino, y luego aplica la función AND con el bitboard de ataque del Rey, que simplemente tiene los ocho bits alrededor de la posición del rey puestos en 1. Los enroques se deberán tratar de forma separada, ya que se debe controlar que no esté impedido de realizar este mo-

vimiento o bien porque se haya movido alguna de las piezas involucradas o porque estén ocupadas los escaques intermedios.

Generador de movimiento de Peones

Los movimientos de los peones también son generados como los del Rey. Los movimientos normales y las capturas necesitan bitboards separados y diferentes condiciones según el estado de ocupación del escaque destino. Las capturas "al paso", también necesitan tratarse por separado porque son capturas con un escaque de destino vacío. Si un peón alcanza la octava o primer fila (según el bando), debemos almacenar las posibles cuatro promociones (Reina, Torre, Alfil o Caballo) como movimientos separados. Es importante mencionar que los peones blancos y negros no pueden compartir los mismos bitboards de ataque porque se mueven en diferentes direcciones.

Generador de movimientos de piezas deslizantes

Los movimientos deslizantes conrresponden a aquellos movimientos realizados por Alfiles, Torres y/o Reinas. Estos movimientos permiten a cualquiera de estas piezas avanzar a cualquier casilla que esté disponible sin importar la distancia de la pieza, siempre que no estén ocupadas las casillas intermedias y sea coherente con el movimiento permitido de la pieza.

La principal diferencia entre este tipo de movimiento y los anteriores es que los bitboards de ataque para estas piezas no solo dependen del escaque de origen, sino que también dependen de la ocupación de las filas, columnas y/o diagonales. Para las piezas no deslizantes, los movimientos pueden almacenarse en una tabla de búsqueda unidimensional. Sin embargo, los bitboards de ataque para piezas deslizantes necesitan almacenarse en tablas de búsqueda bidimensionales. La segunda dimensión se utiliza para denotar la ocupación o estado de la fila, columna o diagonal.

Las piezas deslizantes atacarán una fila, columna o diagonal hasta el final de la misma ó hasta que se encuentre una pieza en dicho camino (sin importar el color de la misma). El atacar una pieza del mismo color no es problema ya que los ataques a piezas de igual color serán removidos cuando se aplique el

AND sobre la negación del bitboard objetivo en base a las piezas blancas (al igual que con piezas no deslizantes). Para el caso de una torre posicionada, por ejemplo en $f1$, el ataque de la primera fila no depende de los contenidos en $a1$ o $h1$ (primer y último bit); el que estos escaques sean atacados depende únicamente de los valores que se tengan en $b1$ y $g1$ y no de ellos mismos. Si no tenemos una pieza en $b1$, el estado de $a1$ será "heredado" de $b1$ y así sucesivamente hasta la posición en la que tengamos ubicada la pieza de estudio. Este método se aplica tanto a filas como a columnas y diagonales. Es interesante observar que para la generación de movimientos de piezas deslizantes, se pueden omitir los bits de ambos extremos y solo necesitaremos los 6 bits internos para definir el estado de ocupación.

Estado de los escaques

El cálculo de la tabla de estado de ocupación de los escaques se realiza básicamente en dos pasos. Primero extraemos el bitboard de ocupación para una sola fila, columna o diagonal (de ahora en más, las llamaremos líneas, de forma indistinta) utilizando bitboards cargados con máscaras que nos dejarán solamente con los bits apropiados de la línea en cuestión. Segundo, con este bitboard de 64 bits, con un máximo de 6 bits seteados, y todos en una única línea, lo transformamos en un índice de búsqueda de 6 bits. La transformación se realiza de forma diferente para filas, columnas y diagonales. Para filas se realiza el siguiente cálculo:

$$6bitstate = (casillerosOcupados \& MASCARAFILA[from]) >> CORRIMIENTOFILA[from]$$ [1]

Los valores $MASCARAFILA[]$, $MASCARACOLUMNA[]$, $DIAGA8H1MASK[]$ y $DIAGA1H8MASK[]$ (los tres últimos se utilizarán más adelante) son vectores cargados con constantes numéricas y su finalidad es dejar con valor en 1 los bits correspondientes a la fila, columna o diagonal que estemos analizando y los restantes bits con valor 0. De forma análoga, el vector $CORRIMIENTOFILA[]$ contiene la cantidad de bits a desplazar la fila al aplicar el corrimiento a la derecha.

[1] "$>>$ " corresponde a la operación "shift right", que desplaza los bits hacia la derecha la cantidad de lugares indicado por el valor almacenado en $CORRIMIENTOFILA[from]$

Para las columnas se agrega un paso más, haciendo la multiplicación de 64 bits con números "mágicos" que se encargan de transformar una única columna a la última fila y al mismo tiempo mantiene el orden de bits, por lo que solamente resta mover el resultado para alinearlo con los seis bits destinados para el estado de ocupación de esa columna. Se tiene un número "mágico" para cada columna:

columna	**número mágico**
a	0x8040201008040200
b	0x4020100804020100
c	0x2010080402010080
d	0x1008040201008040
e	0x0804020100804020
f	0x0402010080402010
g	0x0201008040201008
h	0x0100804020100804

La fórmula resultante sería la siguiente:

$6bitstate = (casillerosOcupados \& MASCARACOLUMNA[from]) * NUMEROMAGICOCOLUMNA[from] >> 57$

Por último, el cálculo de ocupación para diagonales es similar al realizado para columnas; pero utilizando diferentes multiplicadores. Se define la numeración de las diagonales como:

$diaga8h1 = columna + fila - 1$

Tenemos entonces quince diagonales, numeradas del 1 al 15, con la diagonal 8 siendo la más larga, desde $a8$ hasta $h1$. Para direccionar una diagonal particular, solamente se requieren 5 bits, pero, por simplicidad, serán tratados como si tuvieran seis, ignorando el último bit, de forma que todos los bitboards de ataque diagonales tengan el mismo rango de direcciones. A continuación se muestran los números utilizados en estos cálculos:

diagonal	número mágico a8h1
$diaga8h1 = 1, a1$	0x0
$diaga8h1 = 2, a2 - b1$	0x0
$diaga8h1 = 3, a3 - c1$	0x0101010101010100
$diaga8h1 = 4, a4 - d1$	0x0101010101010100
$diaga8h1 = 5, a5 - e1$	0x0101010101010100
$diaga8h1 = 6, a6 - f1$	0x0101010101010100
$diaga8h1 = 7, a7 - g1$	0x0101010101010100
$diaga8h1 = 8, a8 - h1$	0x0101010101010100
$diaga8h1 = 9, b8 - h2$	0x0080808080808080
$diaga8h1 = 10, c8 - h3$	0x0040404040404040
$diaga8h1 = 11, d8 - h4$	0x0020202020202020
$diaga8h1 = 12, e8 - h5$	0x0010101010101010
$diaga8h1 = 13, f8 - h6$	0x0008080808080808
$diaga8h1 = 14, g8 - h7$	0x0
$diaga8h1 = 15, h8$	0x0

Las primeras y las últimas dos diagonales no necesitan multiplicadores porque son sólo uno o dos casilleros, por lo que el ataque a lo largo de estas diagonales no dependen de su ocupación.
La fórmula resultante para el cálculo de las diagonales paralelas a la diagonal a8h1 es la siguiente:

$$6bitstate = (occupiedSquares\&DIAGA8H1MASK[from])* DIAGA8H1MAGIC[from] >> 57$$

Los índices para las diagonales $a1h8$ se calculan de forma análoga, con la fórmula:

$$diaga1h8 = columna + fila + 8$$

numerándolas desde 1 hasta 15, siendo la diagonal número 8 la más larga

Los números "mágicos" se listan a continuación:

diagonal	número mágico a1h8
$diaga1h8 = 1, a8$	0x0
$diaga1h8 = 2, a7 - b8$	0x0
$diaga1h8 = 3, a6 - c8$	0x0101010101010100
$diaga1h8 = 4, a5 - d8$	0x0101010101010100
$diaga1h8 = 5, a4 - e8$	0x0101010101010100
$diaga1h8 = 6, a3 - f8$	0x0101010101010100
$diaga1h8 = 7, a2 - g8$	0x0101010101010100
$diaga1h8 = 8, a1 - h8$	0x0101010101010100
$diaga1h8 = 9, b1 - h7$	0x8080808080808000
$diaga1h8 = 10, c1 - h6$	0x4040404040400000
$diaga1h8 = 11, d1 - h5$	0x2020202020000000
$diaga1h8 = 12, e1 - h4$	0x1010101000000000
$diaga1h8 = 13, f1 - h3$	0x0808080000000000
$diaga1h8 = 14, g1 - h2$	0x0
$diaga1h8 = 15, h1$	0x0

La última fórmula necesaria para calcular las diagonales se detalla a continuación:

$$6bitstate = (occupiedSquares \& DIAGA1H8MASK[from]) * DIAGA1H8MAGIC[from] >> 57$$

Es importante recordar que estamos trabajando con operaciones de bits, y que cada uno de estos bits se corresponde con un escaque particular del tablero el cual tendrá el valor 1 para el caso en que esté ocupado y 0 en caso contrario.

4.3.4. Almacenamiento de movimientos generados

Hacer y deshacer movimientos es una de las tareas más importantes y que no admiten el más mínimo error, ya que afectaría el funcionamiento del programa. Durante las búsquedas, necesitamos almacenar los movimientos generados a medida que vamos descendiendo jugada a jugada por el árbol de análisis. De esta manera podremos encontrar nuestro camino de retorno sin necesidad de volver a generar todos los movimientos nuevamente. La lista de posibles movimientos para la posición actual es parte de la estructura del

tablero, lo que permite que toda la información de búsqueda esté agrupada y disponible siempre que sea necesaria. Además se almacena en otro vector la lista de jugadas que se han realizado hasta el momento y el registro de las capturas al paso y las opciones de enroque disponibles.

Para poder estudiar cada uno de los movimientos posibles, el programa cuenta con un conjunto de artefactos que gestionan el conjunto de movimientos disponibles para un jugador en determinada posición. Cuando se le ordena al generador de movimientos realizar su trabajo, éste recibe un índice que le indica la posición a partir de la cuál deberá comenzar a almacenar los movimientos generados en el buffer de movimientos posibles (recordar que tenemos otro buffer para los movimientos ya realizados). A medida que el generador de movimientos va generando cada movimiento lo va almacenando en la posición indicada por el índice que recibió por parámetro para luego incrementarlo en 1 y continuar con su proceso. Al final de su tarea retornará dicho índice actualizado. Si observamos con detenimiento, podemos notar que el índice con el que fue invocada la función es la posición del buffer en la cual se encontrará el primer movimiento para el jugador de turno, y la posición que retorna la función es la casilla siguiente a la última casilla utilizada para almacenar todos los movimientos posibles de la posición. Con estos dos índices, podemos recorrer cada uno de los movimientos generados y analizar su conveniencia.

En las funciones de búsqueda, el valor de estos índices (inicio y fin de movimientos generados) es de suma importancia por dos razones críticas. La primera, es que gracias a ellos se puede recorrer todos los movimientos posibles para el jugador actual, sin necesidad de invocar nuevamente a la función de generación cada vez que deseemos un nuevo movimiento e ir descartando los movimientos ya analizados. La otra razón, aún más importante, es que al analizar las posibles respuestas de nuestro rival, el generador de movimientos deberá recibir una posición del buffer adecuada (por ejemplo la primer posición libre), de forma tal que los nuevos movimientos generados, correspondientes a las posibles respuestas de nuestro rival, no sobreescriban los actuales.

4.4. Asignaciones de programación

Las asignaciones de programación fueron realizadas en etapas incrementales. La primera asignación consistió en crear pequeñas funcionalidades diferentes que serían luego utilizadas tanto para pruebas como para el estudio de la posición al momento de evaluar el tablero. La segunda etapa fue la implementación de una función para buscar la posibilidad de jaque mate en un número determinado de pasos. La tercer asignación consistió en la implementación de dos algoritmos para la selección del movimiento a realizar. El primer algoritmo consistió en una función de evaluación del tablero para evaluar numéricamente qué tan bien o mal esta. El segundo algoritmo se encargó de la búsqueda de los movimientos a evaluar.

El conjunto de estas funcionalidades es detallado a continuación junto con el conocimiento aprendido en cada una:

4.4.1. Primera Asignación

La primera asignación consistió en agregar un nuevo comando al motor tal que reciba una partida de ajedrez en formato PGN (en un archivo .pgn), y despliegue un reporte en formato de texto con el siguiente contenido: cabezal PGN tal como fue ingresado, seguido de un mensaje claro y asertivo escrito en lenguaje natural respecto a si la partida está terminada (y con qué resultado) o sigue en curso.

Esta primera funcionalidad desarrollada probó ser de una utilidad invalorable al momento de estudiar una posición determinada y por ejemplo corroborar que la función de evaluación retorna los valores esperados. Permitiendonos estudiar fácilmente cualquier posición de la partida tantas veces como queramos sin necesidad de repetir cada uno de los movimientos previos cada vez. Si bien el motor ya contaba con una funcionalidad para cargar partidas en formato FEN, el poder cargar una partida desde un PGN, con notación algebraica standard permite también cargar su histórico de movimientos y lo que es más importante aún, el poder cargar cualquier partida, ya sea propia o bajada de Internet a partir de una posición que nosotros seleccionemos.

Por otro lado, se pedía implementar funcionalidades tales que, tras cada "ply", se desplieguen las siguientes métricas:

Desde la perspectiva del bando al que le toca jugar: ¿Estoy en jaque? ¿Me dieron mate? ¿Estoy ahogado? (Booleanos); ¿Tengo la posibilidad de tomar un peón de mi rival "al paso"? (enumerar jugadas posibles en notación algebraica)

Desde una perspectiva general: ¿Qué enroques están disponibles? ¿Blanco/corto? ¿Blanco/largo? ¿Negro/corto? ¿Negro/largo? (Booleanos); ¿Cuántas jugadas se llevan jugadas sin mover ni tomar ningún peón? (plies div 2); ¿Alguna posición se ha presentado ya dos veces? ¿Cuáles? (desplegar los FEN). Además, evaluación del **material** ¿qué valor material tiene cada bando?; evaluación **espacial** ¿cuántos casilleros (vacíos o no) pueden ser ocupados por cada bando si le tocara jugar ahora?; evaluación de **dinámica** ¿cuántas jugadas disponibles tiene cada bando, si le tocara jugar ahora?

En nuestro motor se implementó un comando para cada una de esas interrogantes, pudiendo ser consultadas luego de realizar un movimiento.

4.4.2. Segunda Asignación

La segunda asignación consistió en la construcción de una funcionalidad que permita la resolución de problemas de mate a cierta profundidad máxima dada. La nueva funcionalidad se especializa en identificar la existencia de secuencias forzosas de mate a una profundidad de juego máxima preestablecida, tal que:

1. Recibe una posición en formato FEN y una profundidad d (entero, positivo), y determina si el jugador al que le toca jugar puede forzar una secuencia de mate en d jugadas. La expresión "secuencia forzosa de mate en d jugadas" debe interpretarse del siguiente modo: si por ejemplo, $d = 2$, el jugador que es "mano" debe realizar una primera jugada tal que luego, a cualquier respuesta del rival, debe darle mate en la jugada siguiente (segunda, $d = 2$). Por supuesto que la jugada

final, de mate, no tiene que ser única; cada respuesta posible del rival puede llevar a que la jugada de mate sea diferente.

2. Devuelve la salida en un archivo PGN con la posición de inicio en el cabezal (campo FEN), una etiqueta adicional que indica si existe mate en la profundidad solicitada (del tipo: Mate-en-d "V/F"), y, en caso de haber encontrado una secuencia de mate, la misma aparece explicitada en la sección de jugadas junto con todas las otras alternativas de mate.

Ejemplos:

FEN de entrada: "1Q6/8/8/8/8/k2K4/8/8 w - - 0 1"
profundidad *d*: 2
PGN de salida:
[Event "Problema de Mate a profundidad máxima fija"]
[White "Mate en 2"]
[Black "Juega el Blanco"]
[Result "1-0"]
[FEN "1Q6/8/8/8/8/k2K4/8/8 w - - 0 1"]
[Mate-en-2 "Verdadero"]

1. Kc3 Ka2 2. Qb2# 1-0
1. Kc3 Ka4 2.Qb4# 1-0

FEN de entrada: "1Q6/8/8/8/8/k2K4/8/8 w - - 0 1"
profundidad *d*: 1
PGN de salida:
[Event "Problema de Mate a profundidad máxima fija"]
[White "Mate en 1"]
[Black "Juega el Blanco"]
[Result "*"]
[FEN "1Q6/8/8/8/8/k2K4/8/8 w - - 0 1"]
[Mate-en-2 "Falso"]
*

Para resolver esta asignación creamos un nuevo comando en el motor, el cual implementa un algoritmo básico utilizando búsqueda exhaustiva. A

continuación se puede ver su pseudocódigo:

```
MateInN?(board, depth) : bool
{
    if IsMate(board)
        then return true
    if (depth = 0)
        then return (false)
    for each move ∈ GetNextMoves(board)
    do {
        MakeMove(board, move)
        result = CheckOppTurn(board, depth)
        UndoMove(board, move)
        if result
            then return (true)
    }
    return (false)
}
procedureCheckOppTurn(board, depth)
{
    for each oppmove ∈ GetNextMoves(board)
    do {
        MakeMove(board, oppmove)
        result = Mate − In − N?(board, depth − 1)
        UndoMove(board, oppmove)
        if !(result)
            then return (false)
    }
    return (true)
}
```

Este algoritmo es muy similar al de búsqueda por poda α-β que veremos en la sección 4.4.3. En este tipo de algoritmos son necesarios mecanismo de corte y poda para evitar que se tornen inviable para valores muy grandes de N, debido a que incrementan de forma significativa el tiempo de ejecución. El mecanismo de corte fue implementado utilizando un temporizador, el cual al llegar al tiempo configurado corta la búsqueda de la función $MateIn - N$

retornado $false$ (es decir que no se encontró una secuencia que termine en mate para cualquier jugada del oponente) y permite ceder el control a la función de evaluación.

Además implementamos un mecanismo de poda para reducir la cantidad de nodos a analizar dentro del árbol de movimientos. Tal que, si partiendo desde un nodo por determinado camino (rama del árbol) no logramos llegar al mate para algún movimiento respuesta del oponente, descartamos todo ese sub-árbol que tiene como raíz nuestra jugada, ya que no tenemos una opción de mate para cada movimiento de nuestro rival. Este criterio ahorra un tiempo significativo evitando la necesidad de analizar todo el grupo de respuestas del oponente cuando encontramos una respuesta para la cual no podemos llegar al mate.

Por otro lado, en esta función de búsqueda de $MateIn-N$, es vital un correcto manejo del buffer de movimientos. Como vimos en la sección 4.3.4, se van almacenando todos los movimientos posibles de cada jugador alternados, con lo cual el buffer crece significativamente a medida que se incrementa la profundidad de búsqueda pudiendo sobrepasar los límites de memoria asignados.

4.4.3. Tercera Asignación

La tercera asignación estaba dividida en dos partes:

Primera Parte:

Implementar una **función de evaluación** de posiciones de ajedrez tal que, dada una estructura interna de datos que refleje la posición, devuelva un valor numérico asociado a la misma. Dicho valor debe reflejar, de forma heurística, la situación de juego representada por dicha posición, o, dicho coloquialmente, refleja qué bando está mejor y qué tanto mejor está.

La estructura interna del cálculo debe tener por lo menos los 3 siguientes términos:

$a1{*}evalMaterial(pos) + a2{*}evalEspacial(pos) + a3{*}evalDinamica(pos)$, donde:

evalMaterial: es la evaluación del **Material** de la posición que apunta a reflejar la diferencia de material entre los bandos. Asigna un valor material parametrizable a cada tipo de pieza ($valD = 9$ para cada Dama, $valT = 5$ para cada Torre, $valA = valC = 3$ para cada Alfil y cada Caballo, y $valP = 1$ para cada Peón) y la evaluación consiste en restarle al valor material total del Blanco el valor material total del Negro.

evalEspacial: es la evaluación **Espacial** de la posición y apunta a reflejar la diferencia en la cantidad de casillas que controlan los bandos. Debe considerar por lo menos aquellas casillas controladas de forma indisputada por un bando y otro.

evalDinámica: es la evaluación **Dinámica** de la posición y apunta a reflejar la diferencia de movilidad existente entre los bandos. Esta evaluación consiste en computar, para cada bando, la cantidad de jugadas legales posibles de sus piezas, multiplicándolas por el valor de las mismas (por ejemplo: si una Torre puede realizar 7 jugadas legales, dicha Torre aportará $7 * valT$ al total de movilidad de su bando).

Segunda Parte

Implementar una técnica de tipo MiniMax, utilizando la función de evaluación de la primera parte para seleccionar las jugadas a recomendar. El motor deberá, adicionalmente, incorporar un parámetro de profundidad de $Mate - In - N$ (funcionalidad vista en la sección 4.4.2) que utilizará durante sus evaluaciones, de modo de detectar si existe alguna línea de mate en profundidad máxima $Mate - In - N$, tanto para usar a su favor si dispone de mate como para evitar jugadas que le permitan dar mate a su rival.

Para la primera parte de esta asignación, comenzamos implementando una función de evaluación aplicada a los nodos hoja del árbol de búsqueda y considerando los tres componentes requeridos: Material, Espacial y Dinámica. La evaluación Material fue implementada en base a los valores asignados a cada una de las piezas involucradas en la partida y para calcular esta métrica restamos al valor material de blancas el valor material de negras. Los valores a las piezas fueron asignados tal como se describieron en el requerimiento (1 para Peones, 3 para Caballos y Alfiles, 5 para Torres y 9 para la Dama).

La segunda métrica requerida fue la evaluación espacial, que permite ana-

lizar cuánto espacio es dominado por cada bando en el tablero. El cálculo se restringe a analizar qué casillas están controladas por cada bando de forma exclusiva. Comienza generando todos los posibles movimientos para el bando que tiene el turno y registra todas las casillas a las cuales se puede acceder. Luego se cede el turno al bando contrario y se repite nuevamente el proceso para finalmente retornar la resta entre los diferentes totales calculados anteriormente.

Finalmente, dentro del grupo de métricas requeridas en la primera parte de esta asignación, tenemos la evaluación dinámica. Tal como se detalló en su definición, apunta a reflejar la movilidad que tienen las piezas de cada bando y ponderarlas de acuerdo al poder ofensivo de la pieza que realiza el movimiento. En nuestro caso por cada movimiento posible cada Peón aportará una unidad a la métrica, los Caballos y Alfiles aportarán tres unidades, la Torre cinco unidades y la Reina nueve. Observemos que cuanto más compacta esté la formación de nuestras piezas, más desfavorable será la métrica, lo cual tiene cierta lógica ya que se estaría reduciendo el poder ofensivo de las piezas.

Después de realizar diferentes pruebas la fución de evaluación y estas métricas, encontramos que el juego de nuestro motor no era lo suficientemente fuerte, por lo que decidimos incluir una métrica más a nuestra función de evaluación. Esta nueva métrica está basada en el análisis de posiciones relativas de las piezas en el tablero y busca bonificar o penalizar determinadas formaciones de piezas. Por ejemplo se puede bonificar el tener ambos Alfiles (en casilleros blancos y negros) o tener cubierto al Rey por un escudo de Peones. Además se puede agregar una bonificación según el tipo de pieza y su posición en el tablero. Por otro lado se puede penalizar cuando se tienen dos o más peones en la misma columna.

Es importante destacar el comportamiento diferenciado de bonificaciones para el Rey según la etapa del juego. Si estamos en etapas iniciales y medias, se bonificará al jugador por mantener su Rey protegido por sus Peones y lejos de piezas enemigas. Sobre el final del juego, se bonificará si el Rey está en posiciones centrales y nuevamente protegido por sus peones.

Como resultado de las purebas y análisis, implementamos la siguiente función de evaluación:
$f(t) = a1 * Material(t) \ + \ a2 * Espacial(t) \ + \ a3 * Dinamica(t) \ + \ a4 *$

$Diferencial(t)$

En la segunda parte de la asignación nos enfocamos en el algoritmo de búsqueda de movimientos. El cual es fundamental dado que el Ajedrez es un juego con una enorme cantidad de movimientos posibles y generalmente para cada jugada que se realiza se tiene un gran número de movimientos de respuesta. Al inicio del juego se tienen veinte movimientos posibles en blancas, y para cada uno de estos movimientos, tendremos veinte movimientos posibles de respuesta en negras (evidentemente, algunos mejores que otros). A medida que se va desarrollando el juego el número de movimientos crece de forma significativa, para luego reducirse sobre el final. Esto lleva a que no sea posible analizar a priori todas las jugadas de una partida de Ajedrez, haciendo imperativo el desarrollo de funciones de búsqueda con procesamiento acotado, para que sea posible llevar a cabo una partida en un tiempo razonable.

Tal como propuso Claude Shannon, en su texto sobre ajedrez de 1950 (Programming a Computer for Playing Chess) [1], es necesario limitar la profundidad de la búsqueda en el árbol de posibilidades y determinar su valor mediante una función heurística. Los dos parámetros más utilizados para acotar la búsqueda son: tiempo y profundidad del árbol de búsqueda. El concepto de acotar una búsqueda por tiempo es bastante simple y natural. Como no parece razonable esperar horas, días o aún más para tener respuesta a una jugada en Ajedrez, fijamos un tiempo razonable que no haga perder la sensación de "interactividad" entre la máquina y el jugador humano (en un entorno de segundos o a lo sumo minutos). El segundo concepto es el de acotar la profundidad del árbol de búsqueda reduciendo la cantidad de jugadas "a futuro" que puede ver nuestro motor, tomando como base su primer movimiento. El árbol de búsqueda se va construyendo de forma alternada con un movimiento nuestro y uno del oponente, por lo que su profundidad está dada por la cantidad de movimientos que hay en esos caminos, que se van creando a medida que alternamos jugadas con nuestro oponente. Es importante destacar que las jugadas del oponente no son realmente ejecutadas por él, sino que son jugadas que el motor supone que realizará el oponente en base a la jugada que esta considerando ejecutar.

En la figura 4.4 se muestra un posible árbol de jugadas para un juego cualquiera. Imaginemos los triángulos con la punta hacia arriba como movi-

mientos nuestros y los de punta hacia abajo como movimientos del oponente. Comenzando desde la parte superior, tenemos que al realizar nosotros el primer movimiento, podríamos tener tres movimientos respuesta diferentes de nuestro rival y a su vez, nosotros responderíamos con alguno de los movimientos del nivel inmediatamente inferior a éste y así sucesivamente. Como ya mencionamos antes, para un inicio de partida de ajedrez, tendríamos veinte nodos iniciales, y para cada uno de éstos, veinte nodos de respuesta, incrementándose la cantidad de nodos a medida que avanza el juego, haciendo imposible el análisis exhaustivo de cada uno de los posibles finales de la partida.

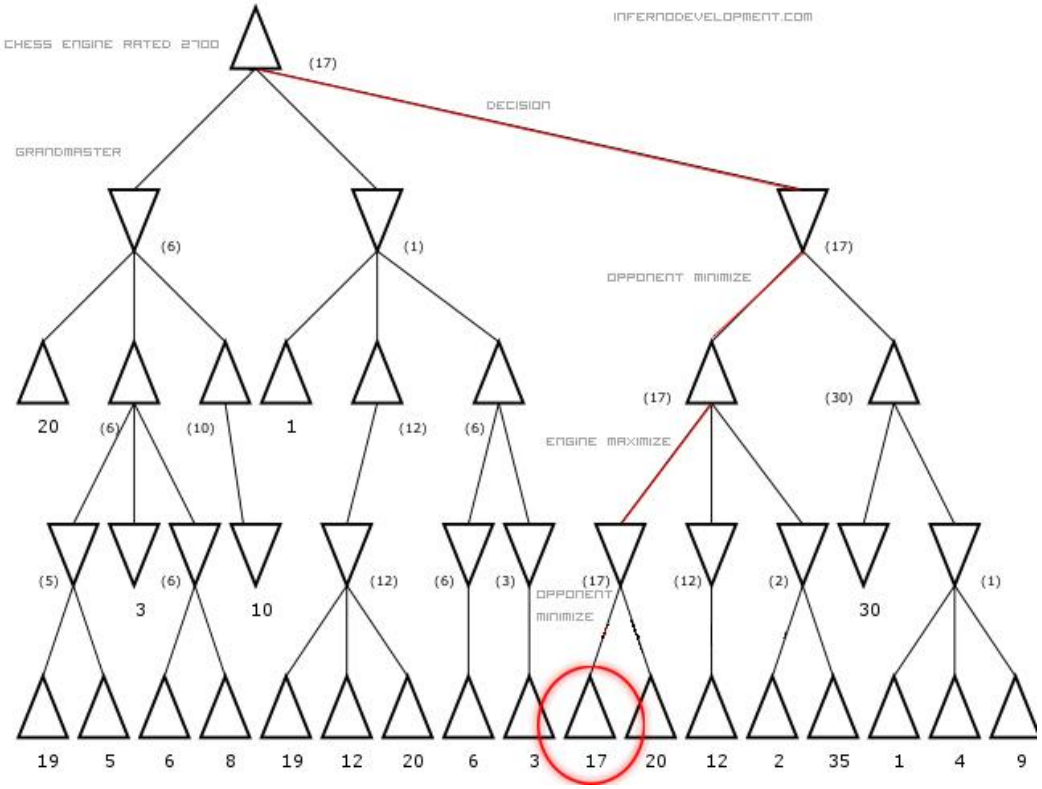

Figura 4.4: Árbol MiniMax de jugadas posibles

Como el lector ya puede haber intuído, la cantidad de nodos en cada nivel aumenta exponencialmente. Aún en profundidades relativamente bajas se tiene una cantidad de nodos fuera del alcance de la capacidad de cómputo

de un computador doméstico disponible al día de hoy.

ALGORITMO MINIMAX

El razonamiento anterior de análisis de posiciones, es el algoritmo "clásico" de búsqueda en Ajedrez, al que si agregamos una función de evaluación cualquiera a aplicarse en cada nodo hoja, obtendremos el algoritmo de búsqueda y evaluación llamado $MiniMax$

Antes de continuar con el algoritmo de búsqueda creemos importante mencionar la **Teoría de juegos** [28], la cual ayudará a comprender las estrategias utilizadas.

La **Teoría de juegos** fue desarrollada en sus comienzos como una herramienta para entender el comportamiento de la economía y actualmente se utiliza en muchos campos, como en la biología, sociología, psicología, filosofía, lógica e informática. La teoría estudia la elección de la conducta óptima cuando los costes y los beneficios de cada opción no están fijados de antemano, sino que dependen de las elecciones de otros individuos.

Los analistas de juegos utilizan la teoría de juegos en conjunto con otras áreas de la matemática, en particular las probabilidades, las estadísticas y la programación lineal. Los juegos estudiados por dicha teoría están bien definidos por objetos matemáticos. Un juego consiste en un conjunto de jugadores, un conjunto de movimientos (o estrategias) disponible para esos jugadores y una especificación de recompensas para cada combinación de estrategias. Por lo tanto, los juegos se pueden representar como árboles, donde cada vértice o nodo representa un punto donde el jugador toma decisiones. Cada nivel del árbol corresponde a un jugador y las líneas que parten del vértice representan las acciones posibles para el jugador. Las recompensas se especifican en las hojas del árbol. Como se puede ver, nosotros nos basamos en esa definición para armar nuestro árbol de búsqueda y así evaluar los movimientos.

La teoría clasifica los juegos en muchas categorías que determinan qué métodos particulares se pueden aplicar para resolverlos. Las categorías que se pueden aplicar a nuestra función de búsqueda son:

- Juegos de $suma - cero$:
  En estos juegos el beneficio total para todos los jugadores, en cada combinación de estrategias, siempre suma cero (en otras palabras, un jugador se beneficia solamente a expensas de otros). El ajedrez es un ejemplo de juegos de $suma - cero$, porque se gana exactamente la cantidad que pierde el oponente.

- Criterios $MaxiMin$ y $MiniMax$:
  Los criterios $MaxiMin$ y $MiniMax$ establecen que cada jugador debe minimizar su pérdida máxima. En el criterio $MaxiMin$ el jugador A, elige que su cobro mínimo posible sea el mayor. Por otro en el criterio $MiniMax$, el jugador B elige que el pago máximo a A sea el menor posible.

- Equilibrio de Nash [29]: Un par de estrategias tienen un equilibrio de Nash, si la elección del jugador A es óptima, dada la elección de B, y la de B es óptima, dada la de A. Este equilibrio puede interpretarse como un par de expectativas sobre la elección de cada persona tal que, cuando la otra revela su elección, ninguna de las dos quiere cambiar de conducta. En otras palabras, el equilibrio de Nash es, en la teoría de juegos, un concepto de solución para juegos de dos o más jugadores el cual asume que hemos elegido nuestra mejor estrategia en función de la estrategia del resto de los jugadores, que son conocidas por todos.

Los motores de ajedrez son capaces de anticiparse a los movimientos de su rival y predecir cuál sera la estrategia que adoptará, para, de esta manera, elegir su siguiente jugada. En general, no sólo el ajedrez si no cualquier juego de estrategia sigue este mismo esquema lógico: adoptar la mejor estrategia teniendo en cuenta las decisiones de su rival (equilibrio de Nash). En este sentido, ningún jugador gana nada modificando su posición si el resto de jugadores no modifican la suya.

Es importante saber que el equilibrio de Nash no implica que se vaya a lograr el mejor resultado para todos los participantes, sino maximizar el beneficio individual posible, teniendo en cuenta que existen dos o más jugadores y que cada uno va a intentar hacer su mejor jugada. Sin embargo, es perfectamente factible lograr un mayor beneficio en conjunto si se toman decisiones coordinadas. Y precisamente la palabra “coordinar” es la clave,

dado que por ejemplo en el dilema del prisionero[2], la acción coordinada de todos los participantes en el juego reduce la condena total de los jugadores. Si ninguno de los dos declaraba en contra del otro, la condena se reducía a seis meses para cada uno, mientras que si cada uno intentaba seguir su propia estrategia, que seguramente fuera la de delatar a su rival para así salir libre, seguramente alguno de ellos tendría que pasar un tiempo largo en la cárcel.

El equilibrio de Nash es una variación de este dilema del prisionero modificado con el fin de resaltar los efectos descritos. En esta versión hay varios jugadores (más de tres). El resultado sería mejor para todos si todos cooperaran entre ellos y no declararan, pero, dado que cada cual persigue su propio interés, y ninguno puede confiar en que nadie declarará, todos deben adoptar la estrategia de declarar, lo que termina en una situación (equilibrio) en la cual cada uno minimiza su posible pérdida.

De hecho, aún cooperando, la estrategia puede ser inestable. Suponiendo que hayan pactado que ninguno de ellos va a declarar, no podemos asegurar que todos ellos vayan a cooperar para reducir la condena conjunta. El poder salir libre de la cárcel puede hacer que muchos jugadores rompan lo pactado y se decidan por declarar. De esta manera, la cooperación no habrá servido para nada.

Ahora que hemos comprendido un poco más la teoría de juegos, describiremos los algoritmos utilizados para nuestra función de búsqueda.

El método base utilizado para nuestra función de búsqueda fue el $MiniMax$ que como ya mencionamos es uno de los criterios de la teoría de juegos. El teorema de $MiniMax$ fue definido por John von Neumann en 1926 [28]. Este teorema establece que en los juegos bi-personales de suma-cero y en los que cada jugador conoce por adelantado la estrategia de su oponente y sus consecuencias, existe una estrategia que permite a ambos jugadores minimizar la pérdida máxima generada.

[2]La policía arresta a dos sospechosos. No hay pruebas suficientes para condenarlos. A ambos se les ofrece el mismo trato pero por separado. Si uno confiesa y su cómplice no, el cómplice será condenado a la pena de 10 años, y el primero será liberado. Si uno calla y el cómplice confiesa, el primero recibirá esa pena y será el cómplice quien salga libre. Si ambos confiesan, ambos serán condenados a 6 años. Si ambos lo niegan, todo lo que podrán hacer será encerrarlos durante seis meses por un cargo menor

Al examinar cada posible movimiento, el jugador debe examinar cada posible respuesta del jugador contrario y la pérdida máxima que puede tolerar. Por lo tanto se utiliza la estrategia de minimizar la pérdida máxima del jugador de turno. La estrategia será óptima para ambos jugadores sólo en el caso que sus MiniMax sean iguales en valor absoluto y opuestos en signos.

Pasos del algoritmo $MiniMax$:

1. Generar todos los nodos del árbol hasta llegar a un nodo terminal, obteniendo el árbol de juego

2. Asignar el puntaje de evaluación a los nodos terminales según la función de evaluación utilizada

3. Calcular el valor de los nodos superiores a partir de los valores inferiores. Dependiendo si es MAX ó MIN se elegirán los valores mínimos o máximos que se obtengan de evaluar la posición movida según se corresponda con los movimientos del jugador o de su oponente.

4. Elegir la jugada basándose en los valores que han llegado hasta el nivel superior.

Con este algoritmo exploraremos todos los nodos del árbol asignándoles un valor numérico mediante la función de evaluación, empezando por los nodos hoja o terminales (los que se encuentran al final del camino a estudiar) y subiendo hacia la raíz. La función de evaluación será la encargada de definir que tan buena o mala es la posición para un jugador cuando la alcanza.

Si bien esta fue la primer aproximación de la función de búsqueda que utilizamos, comprobamos que ya desde profundidades muy bajas de análisis, el tiempo insumido para estudiarlas era por demás excesivo. Por lo tanto, ¡necesitamos más velocidad! y no tenemos más poder de cómputo de donde echar mano.

Si estamos acotados en velocidad de cómputo, la otra opción que tenemos para reducir el tiempo de análisis es reducir el tamaño de la muestra a analizar. El problema es cómo hacerlo. No podemos reducir significativamente la profundidad de búsqueda ya que si lo hacemos, nuestro jugador se hará más “tonto”. Por lo tanto necesitamos reducir la cantidad de muestras

a analizar sin afectar significativamente nuestro juego. Una de estas posibilidades es "podar" el árbol de movimientos que se va generando a medida que bajamos en profundidad, con la idea subyacente de que a menos nodos a estudiar, tendremos más velocidad. El problema que enfrentamos ahora es cómo decidir cuál nodo es lo suficientemente valioso para ser analizado y cuál puede ser descartado junto con sus descendientes para ahorrar tiempo. Esto nos llevó a buscar un reemplazo para nuestra función de búsqueda, que, de alguna forma redujera la cantidad de nodos a estudiar.

Basándonos en la propuesta de Shannon [1] para limitar la profundidad de búsqueda en el árbol de posibilidades, se puede optimizar $MiniMax$ limitando la búsqueda por nivel de profundidad y por tiempo de ejecución. Y además, se puede utilizar la poda α-β. Esta optimización se basa en la suposición que el jugador contrario no nos permitirá jugar nuestras mejores jugadas. Por lo tanto nuestro siguiente paso fue aplicar el algoritmo $MiniMax$ con poda α-β que explicaremos a continuación.

Algoritmo MiniMax con poda α-β [30]

Esta mejora sobre el algoritmo MiniMax es una optimización muy utilizada en los motores de ajedrez y otros juegos con adversarios. Consiste en aplicar una metodología de poda al árbol de búsqueda de forma tal de reducir la cantidad de nodos u hojas a evaluar (inconveniente principal de la búsqueda con el algoritmo $MiniMax$).

La poda α-β parte del hecho que no es posible realizar un análisis exhaustivo a cada camino que se pueda generar en el árbol de búsqueda creado por el algoritmo $MiniMax$ y trata de eliminar grandes partes de éste. Se tratan de eliminar ramas de forma tal que se obtenga el mismo movimiento resultado, pero podando aquellas que no influyan en el resultado final. Como el algoritmo $MiniMax$ realiza primero una búsqueda en profundidad, en cualquier momento, sólo se deben considerar los nodos que forman parte de ese camino en el árbol. La poda α-β, obtiene su nombre de los parámetros α y β que denotan los límites sobre los valores hacia atrás que aparecen a lo largo de cada camino. El valor de α corresponde a la mejor opción hasta el momento (a lo largo del camino) y se le asigna el valor más alto de MAX. El parametro β corresponde a la mejor opción hasta el momento y se le asigna en este caso el valor más bajo MIN. La búsqueda α-β actualiza dichos parámetros a medida que va recorriendo el árbol, realizando la poda de las

ramas restantes cuando el valor que se está examinando sea peor que el valor actual de α o β para MAX o MIN según corresponda.

A continuación podemos ver el pseudocódigo del algoritmo $MiniMax$ con poda $\alpha - \beta$:

```
MiniMaxαβ(depth, α, β) : int
{
    if (depth = 0)
       then return eval()
    for each move ∈ GetNextMoves(board)
    do {
       makeMove(board, move)
       val = −MiniMaxαβ(depth − 1, −β, −α)
       UndoMove(board, move)
       if (val >= β)
          then return β
       if (val > α)
          then α = val
    }
    return α
}
```

La eficacia de la poda $\alpha - \beta$ depende del orden en el que se estudien los nodos sucesores. El algoritmo será más eficiente si se examinan primero los sucesores que probablemente sean los mejores.

Algoritmo MiniMax con poda α-β PVS [32] [33]

La poda PVS (Principal Variation Search) es una mejora a al algoritmo de poda α-β. Nuestra implementación final de la función de búsqueda se basa en esta idea. Una vez que encontramos un movimiento cuyo puntaje se encuentra entre los valores de α y β antes definidos, los siguientes nodos se evaluarán con la meta de probar que son todos peores al que tenemos. Este proceso se puede hacer más rápido ya que vamos reduciendo la ventana creada por α y β realizando aún más podas. Si el algoritmo encuentra que la suposición estaba incorrecta, deberá buscar nuevamente con la ventana de

α-β normal partiendo desde esta nueva Variación Principal.

Solamente necesitamos una revisión leve para darnos cuenta si el movimiento que estamos visitando es inconveniente para nosotros o para el contrario en la mayoría de los nodos y no es necesario contar con un puntaje exacto. Este puntaje exacto es solamente necesario en la llamada PV (Variación Principal), que es una secuencia de movimientos aceptable para ambos jugadores. Lo cual se espera que se propague hasta la raíz del árbol. Si una búsqueda a profundidad baja ha definido cierta secuencia, deberíamos poder encontrar series de movimientos cuyos valores sean mayores que un α y menores que un determinado β a lo largo de toda la rama. Para determinar el puntaje de un nodo de la PV sólo se analiza en su totalidad el primer movimiento, aquel que ha resultado mejor en la iteración previa en un marco de profundización iterativa.

Una vez que tenemos un movimiento en la PV (aquel que haya elevado el valor de α), asumimos que lo elegimos, y para confirmar nuestra elección, centramos una ventana de búsqueda "nula" o cero en este valor de α. Entonces buscamos si hay algún movimiento que puede darnos mejores resultados. Si esto sucede con respecto a la ventana nula, tenemos que realizar una nueva búsqueda con el tamaño de ventana completo. La ventaja principal de esta técnica reside en las búsquedas con ventana nula debido a que son más económicas, y con un buen ordenamiento de movimientos es esperable un ahorro significativo del esfuerzo de búsqueda.

A continuación presentamos un pseudocódigo de esta función:

```
αβpvs(ply, depth, α, β) : int
{
    pvfound = false
    if (depth = 0)
       then return eval()
    for each move ∈ GetNextMoves(board)
    do {
       MakeMove(board, move)
       if (pvfound)
          then{
             val = −αβpvs(ply + 1, depth − 1, − α − 1, − α)
             if ((val > α) && (val < β))
                then val = −αβpvs(ply + 1, depth − 1, − β, − α)
             }
         else val = −αβpvs(ply + 1, depth − 1, − β, − α)
       UndoMove(board, move)
       if (val >= β)
          then return β
       if (val > α)
          then {
             α = val
             pvfound = true
       }
    }
    return α
}
```

4.5. Contratiempos, pruebas y optimización

Contratiempos

El primer problema con el cual nos enfrentamos, y que nos acompañaría en cada una de las pruebas subsiguientes hasta el final del proyecto, fue el tiempo de cómputo insumido para realizar una partida completa, ó retomar alguna partida guardada previamente desde determinada posición, y así poder ver los movimientos que seleccionaba Joli para desarollar el juego. Durante todo el proyecto, contamos únicamente con dos máquinas, cuyos procesadores son un Dual Core de 1.6GHz y un Core i7 de 2.3GHz, ambas con Windows 7. Por lo cual, para jugar una sola partida entre dos versiones diferentes de nuestro motor, o contra el original, se tuvo que esperar desde 40 minutos hasta más de tres horas, dependiendo de la máquina en que se estuviera jugando y los motores que intervenían en la partida.

Otro punto que insumió un tiempo considerable de estudio y pruebas, fue la configuración de las interfaces en donde se jugarían los torneos. La importancia de utilizar una interfaz gráfica radica en que nuestro motor no puede jugar partidas con otro motor de forma directa. Sólo juega contra humanos o contra él mismo (pero con la misma configuración de parámetros), por lo tanto las interfaces son fundamentales para poder realizar torneos entre diferentes configuraciones del motor y así ajustar los parámetros necesarios (tiempo de búsqueda, profundidad, etc) a conveniencia. Comenzamos utilizando dos interfaces gráficas distintas para los torneos: Arena y Winboard. Al poco tiempo nos dimos cuenta que habían ciertos parámetros utilizados por la interfaz que no eran tenidos en cuenta de la forma que nosotros deseábamos. En particular el tiempo de procesamiento asignado a cada motor para la elección de su próxima jugada y la profundidad de la búsqueda. Analizando ambas interfaces, encontramos que este tiempo era variable en cada turno, y muchas veces tenía diferencias considerables entre dos movimientos consecutivos del mismo jugador. Además agregaba segundos dependiendo del estado de la partida. El problema era que las interfaces controlan la profundidad y tiempos asignados a la búsqueda del movimiento ignorando nuestra configuración y generalmente se terminaba el tiempo antes de llegar a la profundidad deseada por nosotros. Esto imposibilitaba realizar pruebas con los valores exactos de configuración que nosotros habíamos elegido. Cabe destacar, además, que los tiempos manejados por la interfaz gráfica, para de-

terminada configuración nuestra, diferían del manejo realizado por el motor ejecutado por línea de comandos con la misma configuración, entorpeciendo aún más nuestras métricas. Visto que no era posible configurar a nuestro gusto ninguna de las interfases de forma directa por su aplicación, comenzamos a buscar caminos alternativos para configurar nuestros motores con la interfaz gráfica. Finalmente encontramos que se podía invocar a la interfaz Winboard desde línea de comandos permitiendo una gran variedad de configuraciones, logrando fijar los parámetros de la configuración de cada uno de nuestros motores para cada competencia que realizamos. De esta manera pudimos fijar el tiempo de búsqueda en un valor elevado tal que permitiera al motor llegar a la profundidad de búsqueda deseada sin ser interrumpido por el temporizador.

Por otra parte, dada la gran cantidad de tiempo insumido para cada partida entre dos versiones de nuestro motor, optamos por no utilizar algoritmos evolutivos en la optimización final de parámetros (profundidad de búsqueda, profundidad de MateIn-N, etc). Para lograr mutar una población inicial significativa se requerirían capacidades de cómputo que exceden de forma considerable a las nuestras, haciendo que los tiempos del proyecto se extendieran de forma descontrolada.

Además de todos los problemas inherentes al desarrollo de un motor de ajedrez en lo que respecta a los tiempos de investigación, desarrollo y capacidad de cómputo, al poco tiempo de haber optado por el motor "Winglet", el sitio donde se encontraban sus páginas de referencia dentro del servidor habían sido dadas de baja. Mucho tiempo después volvimos a encontrar parte de la documentación disponible en la web, pero bajo otro dominio [3].

Pruebas

Dividiremos el proceso de pruebas en dos grandes grupos. El primero de ellos es el grupo de pruebas de funcionamiento del motor por línea de comandos, en el cual partíamos de una posición dada del tablero y analizábamos las valoraciones y decisiones que tomaba el motor en determinadas situaciones. Este tipo de pruebas nos ayudó a estudiar el comportamiento del motor en determinados casos, ayudándonos a detectar malas decisiones del motor que eran potencialmente perjudiciales en vez de elegir opciones más seguras.

El segundo gran grupo de pruebas estuvo formado por interminables horas de torneos entre diferentes versiones de Joli, algunas de ellas con cambios sustanciales en las funciones de valoración de los movimientos y otras con diferentes ponderaciones en las métricas. En este tipo de pruebas, a diferencia de las anteriores, teníamos dos objetivos principales en vez de uno; el primer objetivo de estas pruebas consistía en estudiar cuál motor ganaba la partida, cuyo resultado era extremadamente simple de observar. Si teníamos el mismo motor y dos configuraciones diferentes en la partida, teníamos una imagen de qué métricas hacían aportes más valiosos para el motor. También se debía analizar todo el desarrollo de la partida, ya que podríamos estar enfrentando dos motores, uno de los cuales considerábamos que tendría que presentar un juego más sólido, pero su juego era errático y entregaba piezas que no surtían rédito alguno aunque finalmente terminaba ganando ya que el segundo motor no sabía aprovechar esos errores o directamente presentaba un juego todavía más débil. En este segundo tipo de pruebas fue donde más se sintió el escaso poder de cómputo con el que contábamos ya que el promedio aproximado de tiempo por partida fue de más de una hora. Hubo un caso en particular que demoró más de cinco horas en finalizar.

Es importante destacar que para todas las pruebas realizadas, no se utilizaron libros de apertura, para lograr estudiar el comportamiento del motor desde el inicio de la partida. Además, al tener un libro de apertura, el control del juego podría no cederse a ambos motores al mismo tiempo, por lo que podríamos estar enfrentando un motor contra un libro de aperturas, o adjudicarle un movimiento a un motor cuando en realidad fue realizado por el libro de aperturas.

Optimización

El primer proceso de optimización fue realizado sobre la función de búsqueda de movimientos posibles. Inicialmente, utilizamos el algoritmo MiniMax tal como se había propuesto en la tercer asignación (sección 4.4.3), pero resultó inconveniente debido al excesivo tiempo insumido en analizar posiciones aún a profundidades muy bajas. La causa de este inconveniente, como vimos con anterioridad es el análisis exhaustivo de cada uno de los nodos hoja del árbol de búsqueda. Seguidamente, aplicamos la función de poda $\alpha - \beta$, la cual produjo un descenso significativo del tiempo de análisis, pero aún necesitaba más optimizaciones. El último paso fue aplicar la poda $\alpha - \beta$ PVS, optimizando aún más el procesamiento del árbol de búsqueda.

El hecho de que se insumiera más de una hora en cada partida imposibilitó que se generara una etapa de aplicación de algoritmos genéticos para mejorar cierto conjunto de motores Joli. Con un clúster o equipos realmente potentes, podríamos haber realizado una selección de configuraciones para Joli y de esta forma tener una población suficiente para aplicar alguna técnica de algoritmos evolutivos, para luego obtener una configuración de parámetros óptima para nuestro motor en un tiempo relativamente razonable.

Tomando en consideración la situación a la que nos estábamos enfrentando, optamos por probar determinadas combinaciones de parámetros que pudieran presentar cierto interés desde el punto de vista académico. Con este pequeño conjunto de configuraciones armamos varias versiones de Joli y procedimos a enfrentarlas en torneos, para luego, en base a los resultados ver qué combinaciones eran mejores.

Capítulo 5

Resultados y análisis

Para poder medir el desempeño de nuestra versión final de motor, decidimos iniciar un torneo entre el motor original Winglet y dos instancias de nuestro motor, cada una de ellas con diferentes configuraciones de valoración de métricas. El torneo se desempeñó en las máquinas antes referenciadas (sección 4.5) tomando como base 4 diferentes aperturas. La elección de darle al motor un punto de partida diferente al inicio, se basó principalmente en el hecho de que al tener diferentes arranques, las partidas podrían ser más ricas en cuanto a las elecciones de movimientos. Cabe aclarar que si hubiésemos dejado que los motores eligieran su primer movimiento, al enfrentarlos una y otra vez obtendríamos siempre la misma partida resultado, con la misma secuencia de movimientos. Esto es debido a que la función matemática que evalúa el tablero para una determinada posición retornará siempre el mismo resultado (siempre que se mantengan los parámetros incambiados).

Como mencionamos con anterioridad (en la subsección 4.5), la interfaz gráfica que vinculó a los dos motores en cada una de las partidas fue Winboard y el parámetro de control fue la profundidad de búsqueda, fijada a 8 plys. No aplicamos control de tiempo para permitir a los motores llegar a la profundidad estipulada en cada movimiento.

Antes de entrar en detalle en las pruebas realizadas, detallaremos los parámetros que se pueden configurar en el archivo config.ini de Joli:

- **time**: tiempo máximo de búsqueda (en segundos), nosotros fijamos este valor en 60.

- **depth**: profundidad máxima de búsqueda (en plies), nosotros utilizamos 8

Esos dos parámetros sirven para limitar la búsqueda del movimiento, tal que la búsqueda continúa hasta que se agota el tiempo o se alcanza la profundidad máxima, lo que ocurra primero.

- **STOP_FRAC**: Límite de tiempo para iniciar una nueva iteración; no iniciar una nueva iteración si ya hemos consumido este porcentaje de tiempo de búsqueda.
- **DEPTH_MATE**: Profundidad de búsqueda para la función mateIn-N. el valor por defecto es 2, para deshabilitar la funcionalidad del MateIn-N ingresar el valor 0.
- **TIMER_MATE**: fracción del tiempo de búsqueda de la posición asignado a la búsqueda del mateIn-N, el valor por defecto es 50, si se ingresa 0 entonces se deshabilita la ejecución de esta función
- **EVAL_FUNC**: Parámetro para seleccionar la función de evaluación, si se elije 1, se utilizará la nueva versión implementada por nosotros y si se ingresa 0 se utilizará la del motor original.

Los siguientes parámetros se corresponden con los parámetros $a1$, $a2$, $a3$ y $a4$ (definidos en la sección 4.4.3) utilizados para ponderar la función de evaluación del movimiento:

$f(t) = a1 * Material(t) + a2 * Espacial(t) + a3 * Dinámica(t) + a4 * Diferencial(t)$

- **PARAM_EVAL_MATERIAL**
- **PARAM_EVAL_ESPACIAL**
- **PARAM_EVAL_DINAMICA**
- **PARAM_EVAL_POS_TABLERO**

En base a esos parámetros elegimos dos versiones de Joli que creíamos competitivas, Joli1101 y Joli1001, dónde los números luego del nombre corresponden a los valores de los parámetros detallados en los ítems anteriores en el mismo orden.

Además de la parametrización de la función de evaluación en Joli, se configuró la profundidad de búsqueda de mate en 2. Mediante la interfaz, se otorgó un tiempo suficiente a cada jugada para que los tres motores pudieran llegar a profundidad 8, que fue la preestablecida como tope de búsqueda.

Estas dos instancias de Joli, fueron las dos instancias que a nuestro juicio se desempeñaron mejor en las partidas previas al torneo final. Es importante destacar que la métrica de evaluación dinámica está puesta "a cero", lo que quiere decir que no se está tomando en cuenta para la valoración final del tablero ya que no estaba ayudando a mantener un juego consistente. En particular, al jugar de negras, era propenso a valorar de forma errónea los movimientos iniciales por la inferioirdad relativa de movilidad que se tenía respecto al jugador de blancas, haciendo que se tomaran decisiones muy poco beneficiosas.

En la siguiente tabla, se pueden apreciar los resultados de cada una de las partidas llevadas a cabo entre los tres motores:

movimiento inicial	blancas	negras	resultados
c4-c5	Orig	Joli-1101	0,5 – 0,5
c4-c5	Joli-1101	Orig	1 – 0
d4-d5	Orig	Joli-1101	0 – 1
d4-d5	Joli-1101	Orig	0,5 – 0,5
Nf3-Nf6	Orig	Joli-1101	0 – 1
Nf3-Nf6	Joli-1101	Orig	0 – 1
e4-e5	Joli-1101	Orig	1 – 0
e4-e5	Orig	Joli-1101	1 – 0
c4-c5	Orig	Joli-1001	0,5 – 0,5
c4-c5	Joli-1001	Orig	0 – 1
d4-d5	Orig	Joli-1001	1 – 0
d4-d5	Joli-1001	Orig	1 – 0
Nf3-Nf6	Orig	Joli-1001	0 – 1
Nf3-Nf6	Joli-1001	Orig	1 – 0
e4-e5	Joli-1001	Orig	1 – 0
e4-e5	Orig	Joli-1001	0 – 1
c4-c5	Joli-1001	Joli-1101	1 – 0
c4-c5	Joli-1101	Joli-1001	1 – 0
d4-d5	Joli-1001	Joli-1101	0 – 1
d4-d5	Joli-1101	Joli-1001	1 – 0
Nf3-Nf6	Joli-1001	Joli-1101	0 – 1
Nf3-Nf6	Joli-1101	Joli-1001	0 – 1
e4-e5	Joli-1101	Joli-1001	1 – 0
e4-e5	Joli-1001	Joli-1101	0,5 – 0,5

Como se puede observar en la tabla anterior, ambos motores Joli obtuvieron mejores resultados que el motor original durante el torneo. Lo cual implica una mejora en la capacidad de juego respecto al motor original, cumpliéndose cabalmente los objetivos del proyecto.

Analizando los resultados de las partidas, podemos ver que Joli1001 tuvo un mejor puntaje en las partidas que jugó contra Winglet, ganando el match por tres puntos de diferencia: 5.5 a 2.5. En cuanto a los resultados de Joli1101, si bien ganó el match, sólo lo hizo por dos puntos de diferencia: 5 a 3. Por otra parte, si observamos el desempeño de los motores Joli uno contra el otro, podemos ver que Joli1101 se desempeñó mejor, ganando el match por

3 puntos de diferencia 5.5 a 2.5.

Los puntajes finales del torneo se listan a continuación:

- Joli1101: 10.5
- Joli1001: 8
- Winglet: 5.5

Al finalizar el torneo, es importante destacar la diferencia de puntajes entre Joli1101 y Winglet, donde Joli1101 supera el puntaje de Winglet en casi un 91 %.

A continuación, con ayuda del la herramienta provista por el proyecto de grado "Medición de fuerza de juego por comparación de rendimiento con motores" realizaremos el estudio de las partidas realizadas durante el torneo antes mencionado.

No se consideraron las primeras 12 movidas ya que como mencionamos con anterioridad cuando hablamos de libros de apertura (sección 3.2), éstas jugadas generalmente no condicionan un final seguro y si bien, Houdini al oficiar de "oráculo" puede seleccionar otro movimiento distinto al seleccionado por nuestro motor, no implica necesariamente que el movimiento realizado por Joli sea incorrecto.

En las siguientes gráficas podemos apreciar el resultado del análisis realizado al total de partidas disputadas entre Joli1001 y Joli1101. En el eje de las abscisas tenemos secuencialmente todos los movimientos realizados y el valor promedio de la pérdida de puntaje en base a la diferencia de valoración entre la versión de Joli y el oráculo utilizado, en este caso Houdini. Cuanto más alto es el valor en el eje de ordenadas, más nos estamos desviando de la jugada "óptima" que hubiese elegido el oráculo. En las gráficas puede verse, que si bien Joli1101 tuvo más desviaciones, éstas no fueron tan significativas y la comparación de jugadas "error/perfectas" a nivel del torneo indicaría que Joli1101 es superior a Joli1001. También podemos observar que Joli1101 es superior a Joli1001 en el "mano a mano".

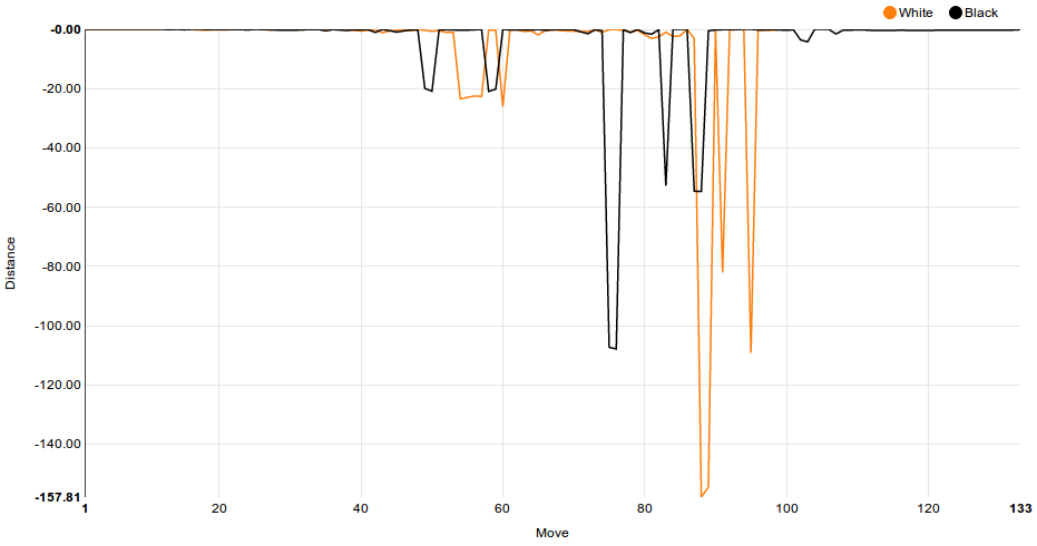

Figura 5.1: Desempeño Joli 1001

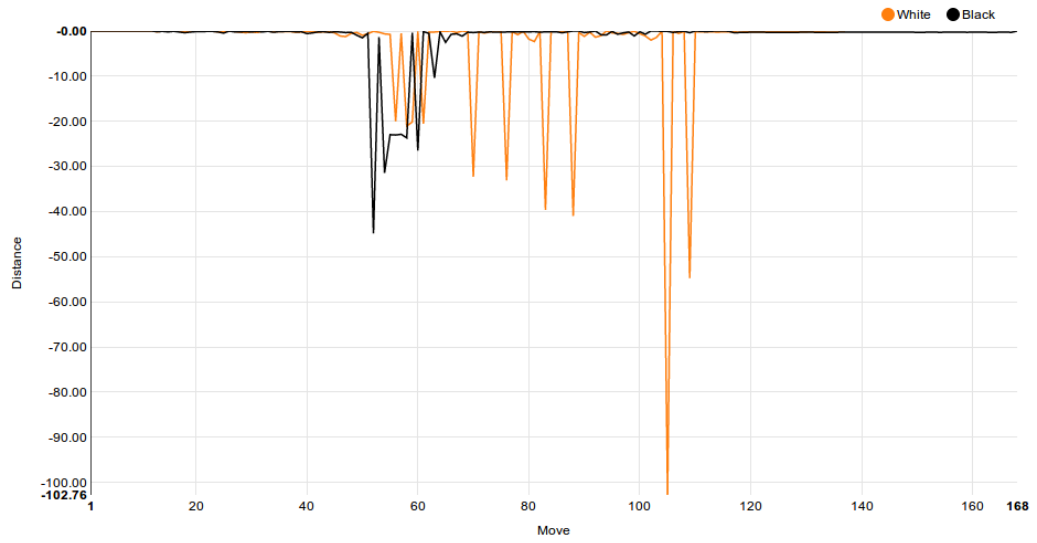

Figura 5.2: Desempeño Joli 1101

En caso que el lector lo desee, en el Apéndice A se pueden encontrar la transcipción de las partidas en formato PGN para este último torneo.

Capítulo 6

Trabajos futuros

Si bien logramos una mejora significativa en el desempeño del motor en comparación con su versión original, existen diferentes técnicas que podrán ser agregadas en trabajos futuros. Algunas de las cuales se listan en las siguientes secciones.

6.1. Tablas de transposición

Muchas posiciones en ajedrez pueden ser obtenidas realizando diferentes secuencias de movimientos, por ejemplo: la posición obtenida por *1.e4 e5 2.f4 d6* es la misma que en la secuencia *1.f4 d6 2.e4 e5*. El obtener posiciones idénticas con secuencias distintas de movimientos se llama transposición. La tabla de tranposición es una tecnica derivada de la programacion dinamica definida por Richard Ernest Bellman (ver sección 3.1)

Cada vez que un programa de ajedrez evalúa una posición, realiza un esfuerzo considerable de cómputo para analizarla, lo que al final se traduce en el consumo de una fracción del tiempo disponible para buscar su mejor movimiento. A cualquier motor de ajedrez le sería de mucha utilidad el poder almacenar la evaluación de dicha posición para usos futuros, de forma tal que la evaluación esté disponible sin tener que recalcularla cada vez que nos topemos con ella.

Como ventajas para este método, podemos citar:

- **Velocidad**: En situaciones donde hay un número significativo de transposiciones, como sobre la etapa final de una partida, la tabla de transposiciones contendrá un alto porcentaje de las posiciones consultadas.
- **Libre profundidad**: Para los casos en los que se necesita buscar cierta posición a determinada profundidad P, si la tabla de transposición posee resultados de $P + K$ movimientos (con $K >= 0$) para la posición inicial, no solo se evita la búsqueda sino que se obtienen mejores resultados que los que se hubiesen obtenido de haber realizado la evaluación.
- **Versatilidad**: La gran mayoría de los programas de ajedrez utilizan libros de apertura. Como la forma de utilizar estos libros es idéntica a como se utilizan las tablas de transposición, podemos inicializar la tabla con el contenido del libro de aperturas de tal forma que cuando llegue una posición que no pertenezca al libro, la tabla de transposición puede contener la información necesaria.

Si bien el aumento de velocidad para analizar posiciones es una ventaja importantísima, este método presenta también un inconveniente no menor, que es su avidez por el consumo de memoria para almacenar los valores y las posiciones estudiadas, ya que la utilidad de una tabla de transposición es mayor cuantos más valores almacene. Por lo tanto es necesario lograr un balance entre memoria disponible y espacio utilizado para la tabla.

6.2. Multithreading

El uso de hilos para estudiar posiciones en ajedrez tiene la ventaja de permitir el análisis de posiciones en paralelo y por consiguiente reducir el tiempo de análisis del total de posiciones a estudiar.

Como toda aplicación que utilice hilos para incrementar su capacidad de procesamiento, un motor de ajedrez debe poner especial énfasis en la sincronización. Una buena práctica para utilizar hilos de forma segura en este tipo de programa es mantener el tablero y sus variables de estado en un ámbito local, de forma que los cálculos de uno de los hilos no sobreescriban alguna variable que precisará otro.

6.3. Optimización del orden de movimientos generados mediante algoritmos evolutivos

Una mejora en el generador de movimientos podría hacerse en lo que respecta al orden en el cual se almacenan los movimientos que esta función va generando. Actualmente los ingresa en un orden predefinido (Peones, Caballos, Alfiles, Torres, Reinas y Rey) sin importar en qué etapa del juego se encuentre. Si estamos en la Apertura, seguramente convenga estudiar el movimiento de peones con mayor detenimiento, pero cuando estamos sobre el final, seguramente convendrá estudiar piezas más ofensivas.

Con algoritmos evolutivos se podría generar un vector de orden que se utilizaría para indicar que los movimientos de determinadas piezas deben tener prioridad sobre otras y por ende, que sean los primeros en estudiar. También podría tenerse en cuenta en qué etapa del juego nos encontramos y en base a ésta, utilizar diferentes vectores de ordenamiento.

6.4. Optimización de funcionalidad $Mate-In-N$ mediante algoritmos evolutivos

Se podría utilizar algoritmos de búsqueda mediante la programación genética para el problema puntual del $Mate - In - N$ para evolucionar una determinada población de individuos. El objetivo es desarrollar un programa que realice búsquedas inteligentes, más que exhaustivas. Por ejemplo, los problemas de $Mate - In - 1$ son típicamente resueltos sólo desarrollando los movimientos de jaque, y no todos los movimientos posibles (los movimientos que no hacen jaque tampoco harán jaque mate y por lo tanto no es necesario analizarlos). Los jugadores humanos a lo sumo consideran 2 ó 3 tableros por segundo, pero de todas formas pueden resolver el problema del $Mate-In-1$ rápido, confiando en el reconocimiento masivo de patrones, poda inteligente o ambos. Se podrían entonces, evolucionar los individuos siguiendo ciertos lineamientos como por ejemplo:

1. Los individuos solamente considerarán movimientos que tengan ciertas

condiciones que ellos mismos han desarrollado por evolución.

2. La profundidad de búsqueda se deja a discreción del individuo con penalizaciones por búsquedas profundas.

3. El desarrollo del árbol de juego es asimétrico, lo que ayuda con el cómputo ya que no necesitamos considerar los mismos aspectos para los movimientos de ambos jugadores.

4. Cada nodo examinado durante la búsqueda es considerado individualmente de acuerdo al conocimiento del juego, y la secuencia de movimientos puede ser desarrollada a una profundidad diferente.

6.5. Optimización mediante algoritmos evolutivos de la función de evaluación

Sean $n = 2^m$ componentes de una función de evaluación para un tablero dado (por ejemplo: evaluación material, espacial, dinámica y posicional). Podemos asociar estos n elementos a un vector con n coordenadas donde la entrada i-ésima sea el "peso" asociado al parámetro i de calidad del tablero.

Podemos entonces corresponder a cada vector de la base canónica con un motor, lo que resultaría en 2^m motores distintos. La idea es tomar parejas al azar de estos motores y hacer una única ronda de 2^{m-1} partidas. Luego quedarnos con la población de 2^{m-1} "ganadores", que serán los ganadores directos de las partidas anteriormente jugadas, junto con un promedio de características de aquellos partidos que hayan terminado en tablas. Por lo tanto la población de cada generación se conformará entonces de los 2^{m-1} individuos ganadores de la generación anterior y los restantes "al azar", sorteando variables aleatorias uniformes en cada coordenada.

Luego de tener nuevamente el total de la población elegida, con 2^m integrantes volvemos a repetir el procedimiento de evolución enfrentando estos motores en parejas y eligiendo los victoriosos.

Para resolver las asimetrías de blancas y negras, se podrían realizar dos partidas por cada enfrentamiento de motores, una de blancas y otra de negras para cada motor, y en caso de empate o igual puntaje, recombinar los

genes promediando los vectores.

También podría incluirse mutaciones genéticas en la población de estudio, mediante alguna función de recombinación o simplemente de forma manual por parte de los desarrolladores, además de utilizar alguna forma más sofisticada de recombinación de genes para la genética de los motores victoriosos.

Es importante mencionar que el tiempo insumido en tomar una población y evolucionarla durante varias generaciones puede ser considerablemente largo, además de requerir un poder de cómputo considerable.

6.6. Metaheurística GRASP

Una alternativa al desarrollo de algoritmos evolutivos es GRASP [34] (Greedy Randomized Adaptive Search Procedure), enriquecido con una técnica de post-optimización conocida como Path Relinking. GRASP es una metaheurística introducida por los doctores Celso Ribeiro y Mauricio Resende, con aplicación en una amplia gama de problemas de optimización combinatoria.

La metaheurística GRASP es multi-comienzo y utilizada en problemas combinacionales, donde cada iteración consiste básicamente en dos fases: construcción y búsqueda local. En la fase de construcción se genera una solución factible y se investiga el vecindario hasta encontrar un mínimo durante la etapa de búsqueda local. La mejor solución global será la que se mantenga como el resultado.

Para el diseño de Grasp es imprescindible: definir una noción greedy del problema, introducir aleatoriedad para agregar diversidad a la heurística, pudiendo explorar nuevas soluciones y definir una estructura de vecindad, en lo posible transitiva, de forma tal de que dadas dos soluciones, sea posible llegar desde una hasta la otra mediantes soluciones vecinas. Esta estructura de vecindad permitiría definir una búsqueda local, y gracias a su transitividad, enriquecerse mediante una técnica de post-optimización como puede ser el Path-Relinking, sugerida por los mismos creadores de GRASP.

Una posibilidad de GRASP podría hacerse con un criterio greedy por

material; la aleatoriedad dada por puntos distribuidos uniformemente en el hipercubo, con la estructura de vecindad como norma 1 en este hipercubo, y definiendo un grafo con puntos explorados y distancia. Los bloques elegidos se podrían construir generando "caminos aleatorios" dentro del hipercubo. Por ejemplo, se podrían sortear n puntos en el hipercubo y enfrentarlos con este motor greedy. Se podría comenzar eligiendo un α dentro del intervalo [0..1], y la lista de candidatos restringida (LCR) igual al conjunto de $n * \alpha$ mejores soluciones de las n soluciones. Luego, sortearíamos al azar la nueva solución dentro de la LCR, y nos desplazamos a la nueva solución, repitiendo nuevamente el proceso. También se podría definir un bloque de búsqueda local, definiendo un grafo con los puntos del hipercubo de la siguiente forma: dado un radio de entrada r, con $r > 0$ (bajo la norma 1), considerar los vértices como todos los puntos visitados, y que cada vértice comparta aristas con aquellos vértices con la distancia entre ellos menor que r. Luego, con el grafo obtenido, aplicamos una búsqueda local.

Todo el proceso anterior podría repetirse hasta conseguir un conjunto S de soluciones $\{S_1..S_t\}$ para luego utilizar una post-optimización como puede ser path-relinking. Para cada par de miembros del conjunto S construiríamos un camino simple entre ellos y se exploraría la calidad del mismo; es decir, la calidad de todas las soluciones que lo componen, sustituyendo las soluciones de ambos extremos por la mejor de todo el camino y repitiendo este proceso hasta conseguir una única solución.

Capítulo 7

Conclusiones

El resultado final de este proyecto es un motor de ajedrez que hemos nombrado "***Joli***", el cual, partiendo del motor base *Winglet* a principios del mes de Abril, logró mejorar su desempeño de forma significativa aplicando nuevos algoritmos de búsqueda, evaluación y ponderaciones a las diferentes componentes de valoración para una posición, cumpliendo cabalmente tanto con los objetivos del proyecto como con las expectativas de nuestro tutor, Dr. Ing. Pablo Romero, y de nuestro usuario responsable, Ing. Gonzalo Varalla.

El desarrollo del motor se basó en la aplicación de las teorías clásicas de programación de motores de ajedrez de Shanon con agregados de parametrización externa a las valoraciones de forma de permitir cambios de estilo de juego sin necesidad de recompilación de código y hasta durante una misma partida[1].

Joli fue afinado durante sucesivos torneos hasta llegar a su parametrización actual de coeficientes de evaluación que lo hacen un rival interesante para el jugador humano promedio, aunque no llegue a un nivel de juego internacional en competencias de motores.

En el torneo final, al enfrentar las dos versiones más fuertes de *Joli* contra su versión original (*Winglet*), resultaron ambas victoriosas por un margen considerable, dándonos la confianza para afirmar que el objetivo principal del

[1]El cambio de parametrizaciones durante una partida solamente es posible cuando se está ejecutando por línea de comandos si modificamos el archivo "config.ini" y luego invocamos el comando "ini".

proyecto, de tomar un motor base, mejorarlo y optimizarlo fue cumplido. En este punto, fuimos ayudados a analizar las partidas por el proyecto de evaluación de rendimiento que generosamente estudiaron los archivos PGN que contenían las partidas y nos dieron un análisis corroborando lo antes dicho.

También quisiéramos acotar, que si bien este proyecto está finalizando, hay grandes oportunidades de mejorar el motor, como ya detallamos en el capítulo de trabajos futuros.

Bibliografía

[1] Claude Shannon: "Programming a Computer for Playing Chess". *Philosophical Magazine*, Ser.7, Vol. 41, No. 314 - Marzo 1950 http://vision.unipv.it/IA1/ProgrammingaComputerforPlayingChess.pdf

[2] B. Jack Copeland, Diane Proudfoot: "Alan Turing - Father of the modern Computer". The Rutherford Journal Volume 4 http://www.rutherfordjournal.org/article040101.html. 2011-2012

[3] Stef Sluijten: "Writing a chess program in 99 steps" http://web.archive.org/web/20120113074155/http://www.sluijten.com/winglet/ http://www.sluijten.com/winglet/index.htm, Septiembre 2011

[4] Tim Mann & H.G.Muller: "Chess Engine Communication Protocol" http://www.gnu.org/software/xboard/engine-intf.html, Septiembre 2009

[5] Tim Mann: "Tim Mann's Chess Pages" http://www.tim-mann.org/chess.html

[6] Edward Winter: "The Chess Historian H.J.R. Murray" http://www.chesshistory.com/winter/extra/murray.html

[7] Raúl Alvarado Díaz: "Ajedrez, Ajedrecistas y la vida". Ed. Atelí Ltda. 1995

[8] Federación Internacional de Ajedrez: Notación algebraica
http://www.feda.org/pdf_arbitros/leyes.pdf. FIDE, Julio 2009

[9] Federación Internacional de Ajedrez: Página oficial
http://www.fide.com/

[10] Bienestar Universitario - Seminario de Ajedrez, Oct 2013
http://www.bienestar.edu.uy/noticias/seminario-de-investigaci%C3%B3n-en-ajedrez
http://www.bienestar.edu.uy/noticias/un-proyecto-que-no-se-detiene-seminario-de-investigaci%C3%B3n-en-ajedrez-en-la-udelar

[11] Universidad de Vigo, España
http://www.uvigo.es/uvigo_es/index.html

[12] Tom Standage: "The Turk: The Life and Times of the Famous Eighteenth-Century Chess-Playing Machine", Berkley Trade, 2003

[13] José García Santesmases: "Obra e Inventos de Torres Quevedo". Instituto de España en la Colección Cultura y Ciencia, 1980.

[14] Norbert Wiener: "Cybernetics, a New Science, Seeks the Common - Elements in Human and Mechanical Brains", MIT Press, 1965
http://home.swipnet.se/allez/Eng/Wiener48.htm

[15] Jorge Egger: "Desarrollo de la Maestría Ajedrecística Computacional utilizando recursos restringidos". Santiago de Chile, Universidad de Chile, Julio 2003. Sección de partidas comentadas
http://users.dcc.uchile.cl/~jegger/memoria/ http://users.dcc.uchile.cl/~jegger/memoria/partidas/partida1.htm

[16] J. J. O'Connor y E. F. Robertson: "Richard Ernest Bellman". University of St Andrews, Scotland, Diciembre 2005
http://www-history.mcs.st-and.ac.uk/Biographies/Bellman.html
https://chessprogramming.wikispaces.com/Richard+E.+Bellman

[17] Richard Bellman: “On the Application of Dynamic Programming to the Determination of Optimal Play in Chess and Checkers”, Publicación Académica, 1966
http://www.mathunion.org/ICM/ICM1966.1/Main/icm1966.1.0065.0082.ocr.pdf

[18] IBM Corp.: Deep Blue Overview
http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/

[19] Graham Banks, Ray Banks, Sarah Bird, Kirill Kryukov y Charles Smith: “Computer Chess Rating List (CCRL)”, Inglaterra, 2006
http://computerchess.org.uk/ccrl/

[20] Ami Hauptman y Moshe Sipper: “Evolution of an Efficient Search Algorithm for the Mate-In-N Problem in Chess”, EuroGP, 2007
http://www.cs.bgu.ac.il/~sipper/papabs/gpsearch.pdf

[21] Comunidad Chessgames.com: “Learning Fosyth-Edwards Notation”
http://www.chessgames.com/fenhelp.html

[22] Steven J. Edwards, ICC Help: “PGN-spec”, 1994
http://www6.chessclub.com/help/PGN-spec

[23] Laws of Chess: “the PGN-notation”, 2003
http://www.mychess.de/ChessNotation.htm

[24] Rudolf Huber y Stefan Meyer-Kahlen: “UCI Protocol ”, 2000
http://www.shredderchess.com/chess-info/features/uci-universal-chess-interface.html

[25] Stefan Meyer-Kahlen: “Shredder Ches”
http://www.shredderchess.com/

[26] Sitio oficial Visual Stuido
http://www.visualstudio.com/es-es

[27] Repositorio de código fuente: GIT
http://git-scm.com/

[28] J. Von Neumann: "Theory of Games and Economic Behavior". Princeton University Press, New Jersey, Morgenstern, O. 1944.

[29] J. F. NASH, JR.: "Equilibrium points in n-person games", PNAS, 1950, vol. 36 no. 1 páginas 48-49.
http://www.pnas.org/content/36/1/48.full.pdf+html

[30] Jorge Valver Rebaza, Pedro Shiguihara Juárez, Juan Grados Vásquez: "Algoritmo de búsqueda Minimax con poda Alfa/Beta y búsqueda Quiescense para el juego del Ajedrez", Escuela de Informática de la Universidad Nacional de Trujillo, Perú
http://joshi.ueuo.com/archivos/Ajedrez.pdf

[31] Jorge Alvarado Valderrama y Lesly Rodríguez Pinillos: "Aplicación del algoritmo poda $\alpha - \beta$ para la implementación del juego Ajedrez",
http://www.seccperu.org/files/APLICACI\%C3\%93N\%20DEL\%20ALGORITMO\%20PODA\%20ALPHA.pdf

[32] Autor desconocido: "Computer Chess Programming - PVS Pruning", 1997
http://chess.verhelst.org/1997/03/10/search/#principal

[33] Bruce Moreland: "Principal Variation Search An enhancement to alpha-beta", 2002
http://web.archive.org/web/20070809015901/www.seanet.com/~brucemo/topics/pvs.htm

[34] Mauricio G.C. Resende y Celso C. Ribeiro: Greedy Randomized adaptive search procedures, 2002
http://www2.research.att.com/~mgcr/doc/sgrasp-hmetah.pdf

[35] Editor de LaTeX https://www.writelatex.com

[36] Referencias LaTeX (1)
http://latexlive.files.wordpress.com/

[37] Referencias LaTeX (2)
http://en.wikibooks.org/wiki/LaTeX/

[38] Programación de Ajedrez - Wikispaces
http://chessprogramming.wikispaces.com/Recommended+Reading

[39] Mejoras en la función de búsqueda para ajedrez Cheoss [chess engine] http://www1.freewebs.com/cheoss/mejoras.html

Apéndice A

Partidas del torneo final

A continuación se listan las partidas del torneo final entre los motores Joli1101, Joli1001 y Winglet en formato PGN.

A.1. Joli1101 vs Winglet

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* " − "]
[*White* "*winglet*"]
[*Black* "*Joli* − 1101"]
[*Result* "0 − 1"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,002... − 0,28"]

1. Nf3 Nf6 2. d4 d5 3. Nc3 e6 4. Bg5 h6 5. Bxf6 Qxf6 6. a3 Nc6 7. e4 dxe4 8. Nxe4 Qg6 9. Qd3 f5 10. Qb5 a6 11. Ne5 Qh7 12. Qa4 fxe4 13. Bb5 Bd7 14. Bxc6 bxc6 15. Qb3 Rc8 16. O-O Bd6 17. Rae1 Qf5 18. Qc3 Rd8 19. Nxc6 Rc8 20. f3 Qd5 21. Nb4 Qh5 22. f4 O-O 23. Rxe4 c5 24. Nxa6 Bb5 25. Qb3 c4 26. Qf3 Qxf3 27. Rxf3 Bxa6 28. Rxe6 Rcd8 29. g3 Kf7 30. Re1 Rb8 31. b4 cxb3 32. Rxb3 Rfc8 33. Reb1 Rxb3 34. Rxb3 Rxc2 35. Rb6 Rc1+ 36. Kf2 Rc2+ 37. Kg1 Bxf4 38. gxf4 Bd3 39. Rb3 Be4 40. Re3 Bc6 41. Rd3 Rg2+ 42. Kf1 Rxh2 43. Rc3 Bb5+ 44. Kg1 Rd2 45. Rc5 Bd7 46. Rc4 Ke6 47. Kf1

Kf5 48. Ke1 Rd3 49. Ke2 Ke4 50. d5+ Kxd5 51. Kxd3 Bb5 52. a4 Bxc4+ 53. Ke3 g6 54. a5 h5 55. Kf3 Kd4 56. f5 g5 57. Kf2 Ke5 58. Ke3 Kxf5 59. Kd4 h4 60. Ke3 Kg4 61. Kf2 h3 62. Ke3 h2 63. Kd4 Bf1 64. Kd5 Be2 65. Ke4 h1=Q+ 66. Kd4 Kf5 67. Kc3 g4 68. a6 Bxa6 69. Kb4 Qb1+ 70. Kc3 g3 71. Kd2 Qb2+ 72. Kd1 g2 73. Ke1 g1=Q#
Checkmate 0-1

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* "−"]
[*White* "*Joli* − 1101"]
[*Black* "*winglet*"]
[*Result* "0 − 1"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,022... − 0,26"]

1. Nf3 Nf6 2. d4 d5 3. Nc3 e6 4. e3 Bd6 5. Bb5+ Bd7 6. O-O Bxb5 7. Nxb5 O-O 8. Nxd6 Qxd6 9. Qd3 Nc6 10. c4 e5 11. Nxe5 Nxe5 12. dxe5 Qxe5 13. f4 Qe7 14. Bd2 Qe4 15. Qxe4 Nxe4 16. Ba5 b6 17. Be1 dxc4 18. Rc1 b5 19. Ba5 c5 20. Rfd1 f5 21. Bc7 Rae8 22. Be5 Rf7 23. h3 h5 24. Kf1 Kf8 25. Ke2 Ke7 26. h4 Ke6 27. Rc2 Rb7 28. Kf3 Kf7 29. Re2 Ke6 30. Ree1 b4 31. Re2 b3 32. a3 Ree7 33. Ree1 Rbd7 34. Ke2 a5 35. Rg1 Rf7 36. Rge1 Rfe7 37. Rg1 Rf7 38. Rge1 Rd5 39. Rxd5 Kxd5 40. Rd1+ Ke6 41. Rd8 Rb7 42. Re8+ Kf7 43. Rd8 c3 44. Rd1 cxb2 45. Bxb2 Ke6 46. Rd8 c4 47. Re8+ Kf7 48. Re5 c3 49. Rxf5+ Kg6 50. Rg5+ Nxg5 51. Bxc3 b2 52. Bxb2 Rxb2+ 53. Kd3 Nh7 54. g4 hxg4 55. f5+ Kxf5 56. e4+ Ke5 57. Kc3 Rh2 58. Kd3 g3 59. Ke3 Ra2 60. Kd3 g2 61. Kc4 g1=Q 62. Kb5 Rb2+ 63. Kc6 Qg4 64. Kc7 Nf6 65. a4 Qd7#
Checkmate 0-1

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,06"]
[*Round* "−"]
[*White* "*winglet*"]
[*Black* "*Joli* − 1101"]
[*Result* "1 − 0"]

[*TimeControl* “300”]
[*Annotator* “2. + 0,052... − 0,39”]

1. e4 e5 2. Nf3 Nf6 3. Nc3 Nc6 4. Bc4 Nxe4 5. Nxe4 d5 6. Bd3 f5 7. Bb5 fxe4 8. Nxe5 Qf6 9. d4 Be7 10. Bxc6+ bxc6 11. O-O O-O 12. f3 Ba6 13. Re1 Rad8 14. Be3 exf3 15. Qxf3 c5 16. Qxf6 Bxf6 17. Rad1 Rfe8 18. c3 cxd4 19. Bxd4 c5 20. Nc6 cxd4 21. Nxd8 Rxd8 22. cxd4 Rc8 23. Rd2 Bg5 24. Rdd1 Rc2 25. Re8+ Kf7 26. Re5 Bf6 27. Rxd5 Be2 28. Rb1 Ke6 29. Ra5 Bxd4+ 30. Kh1 Bxb2 31. Rxa7 Bd4 32. Ra8 Bc4 33. Re1+ Kd7 34. Ra4 Bf2 35. Rd1+ Ke6 36. a3 Bd5 37. Rg4 g6 38. a4 Ra2 39. h3 Bc6 40. Rf1 Ke5 41. Rg5+ Ke6 42. a5 Bh4 43. Rg4 g5 44. Rc1 Bd5 45. Rg1 Ke5 46. Rb4 Ra3 47. Rd1 Rxh3+ 48. Kg1 Re3 49. Kf1 Re4 50. Rxe4+ Kxe4 51. Ke2 h6 52. Rb1 Kf4 53. Kf1 g4 54. Rc1 Bf6 55. Rc5 Ke4 56. a6 Bd4 57. Rc7 Kf5 58. a7 Be4 59. g3 Bd3+ 60. Ke1 Be4 61. Rf7+ Kg5 62. Rf4 Bc3+ 63. Ke2 Bc6 64. Rc4 Bf3+ 65. Kd3 Bb7 66. Rxc3 Kf5 67. Rc8 Ke5 68. a8=Q Bxa8 69. Rxa8 Kf5 70. Rg8 h5 71. Ke3 Kf6 72. Kf4 Ke7 73. Rh8 Ke6 74. Rxh5 Kd6 75. Kxg4 Ke6 76. Kf4 Kd6 77. Ke4 Kc6 78. Kd4 Kd6 79. Rh6+ Kc7 80. Kd5 Kb7 81. Rc6 Ka7 82. g4 Kb8 83. g5 Ka7 84. Kd6 Kb7 85. g6 Ka8 86. Rc7 Kb8 87. g7 Ka8 88. g8=Q#
Checkmate 1-0

[*Event* “*ComputerChessGame*”]
[*Site* “*ALICE*”]
[*Date* “2013,11,06”]
[*Round* “ − ”]
[*White* “*Joli* − 1101”]
[*Black* “*winglet*”]
[*Result* “1 − 0”]
[*TimeControl* “300”]
[*Annotator* “2. + 0,092... − 0,35”]

1. e4 e5 2. Nc3 Nf6 3. Nf3 Bb4 4. a3 Bxc3 5. dxc3 O-O 6. Bg5 d6 7. Qd3 h6 8. Be3 Bg4 9. O-O-O Nbd7 10. h3 Bxf3 11. gxf3 Qe7 12. Qb5 b6 13. Rg1 h5 14. Bh6 Ne8 15. Qc6 Rd8 16. Bg5 Ndf6 17. Bc4 a5 18. Rge1 Qd7 19. Qxd7 Rxd7 20. f4 exf4 21. Bxf4 c6 22. Bb3 d5 23. e5 Nh7 24. c4 Nc7 25. cxd5 cxd5 26. Be3 b5 27. a4 Re8 28. Bb6 Ng5 29. Rd3 Nce6 30. h4 Nf4 31. hxg5 Nxd3+ 32. cxd3 Re6 33. Bd4 bxa4 34. Bxa4 Rd8 35. Bd1 g6 36. Kd2 Kf8 37. Ke3 Kg7 38. Bf3 Rc6 39. Bc3 Ra6 40. Kd4 Ra7 41. Ra1 a4 42. Ra3 Kf8 43. Bxd5

Rad7 44. Bb4+ Kg7 45. Bd6 Rc8 46. Bc4 Rb8 47. Kc3 Rh8 48. Rxa4 h4 49. Bb5 Rb7 50. Bc6 Rb6 51. Bd5 Rb5 52. Bc4 Rb7 53. Bd5 Rd7 54. Bc6 Rc8 55. Ra6 Rdd8 56. d4 h3 57. Kd2 h2 58. Bh1 Rc4 59. d5 Rd4+ 60. Ke2 Rg4 61. Be7 Rb8 62. Bf6+ Kg8 63. e6 Rg1 64. Ra3 fxe6 65. Rh3 Rb6 66. Rxh2 e5 67. Rh8+ Kf7 68. Bxe5 Rxg5 69. Bd4 Rb3 70. Rh7+ Ke8 71. d6 Rb8 72. Bc6+ Kf8 73. Rh8+ Kf7 74. Rxb8 Rh5 75. d7 Ke6 76. d8=Q Re5+ 77. Bxe5 Kxe5 78. Qe7+ Kf5 79. Bd7+ Kf4 80. Rb4#
Checkmate 1-0

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,06"]
[*Round* "−"]
[*White* "*winglet*"]
[*Black* "*Joli* − 1101"]
[*Result* "0 − 1"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,012... − 0,28"]

1. d4 d5 2. Nc3 Nc6 3. e4 e6 4. Bb5 dxe4 5. Bxc6+ bxc6 6. Nxe4 Qd5 7. Qf3 Bb7 8. Be3 O-O-O 9. a3 Qa5+ 10. Bd2 Qb6 11. Qxf7 Qxb2 12. Qxe6+ Kb8 13. Ra2 Qb1+ 14. Ke2 Qb5+ 15. Ke1 c5 16. Nc3 Re8 17. Nxb5 Rxe6+ 18. Ne2 a6 19. Nxc7 Kxc7 20. c4 cxd4 21. Kf1 Bc5 22. Kg1 Nf6 23. Nf4 Rc6 24. a4 Ne4 25. Ba5+ Kd7 26. Nd3 Bd6 27. Rb2 Rb8 28. Rc2 Nc5 29. Nb4 Rcc8 30. f3 Nb3 31. c5 Nxc5 32. Na2 Nd3 33. Rd2 Bd5 34. Rxd3 Rb1+ 35. Kf2 Rc2+ 36. Rd2 Rxd2+ 37. Bxd2 Rxh1 38. Nc1 Rd1 39. Ke2 Rh1 40. Kf2 Bxh2 41. a5 Bg1+ 42. Kg3 Bc4 43. Kf4 Be3+ 44. Bxe3 dxe3 45. Ne2 Bxe2 46. Kxe3 Bb5 47. Kf4 Bc6 48. Kg4 Ra1 49. Kf4 Ra2 50. Kg3 Bb5 51. Kh3 Bf1 52. Kg4 Bxg2 53. Kg3 h5 54. Kf4 h4 55. Kg4 h3 56. Kg3 Bf1 57. f4 h2 58. Kg4 h1=Q 59. Kf5 Bd3+ 60. Kg4 Rg2+ 61. Kf3 Qh3#
Checkmate 0-1

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,06"]
[*Round* "−"]
[*White* "*Joli* − 1101"]
[*Black* "*winglet*"]

[*Result* "1/2 − 1/2"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,022... − 0,27"]

1. d4 d5 2. Nc3 Nc6 3. Nf3 e6 4. e4 Bb4 5. e5 Bxc3+ 6. bxc3 Nge7 7. Bd3 O-O 8. O-O Bd7 9. Rb1 Rb8 10. c4 a5 11. Bd2 f6 12. Re1 fxe5 13. dxe5 dxc4 14. Bxc4 Qe8 15. Bg5 h6 16. Bd2 b6 17. Qe2 Qf7 18. Bd3 Nf5 19. Qe4 Nce7 20. Qc4 Bc6 21. Be4 Bxe4 22. Qxe4 Rbd8 23. Kh1 Qe8 24. Kg1 Qc6 25. Kf1 g5 26. Rbd1 Qxe4 27. Rxe4 Nd5 28. Ke2 Rfe8 29. g4 Ng7 30. c4 Nb4 31. a3 Na6 32. Rd4 Rxd4 33. Nxd4 Kh7 34. h4 gxh4 35. Rh1 Kg6 36. f3 h5 37. Rxh4 hxg4 38. Rxg4+ Kf7 39. Rh4 Nc5 40. Nb5 Re7 41. Be3 Nb3 42. Bf4 Rd7 43. Be3 Kg6 44. Rg4+ Kf7 45. Rh4 Kg6 46. Rg4+ Kf7 47. Nd4 Nxd4+ 48. Bxd4 Nf5 49. Bc3 c5 50. Rg2 Ke7 51. Rh2 Ng3+ 52. Kf2 Nf5 53. Ke2 Ng3+ 54. Kf2 Nf5 55. Rh7+ Kd8 56. Rxd7+ Kxd7 57. a4 Nd4 58. Ke3 Nf5+ 59. Ke4 Ng3+ 60. Kd3 Nf5 61. Ke4 Kc6 62. Bd2 Ng3+ 63. Kd3 Nf5 64. Bg5 Ng3 65. Bf4 Nf5 66. Ke4 Nd4 67. Be3 Ne2 68. Kd3 Ng3 69. Bf4 Nf5 70. Ke4 Kd7 71. Bd2 Kc6 72. Bc3 Kd7 73. Kf4 Kc7 74. Ke4 Kc6 75. Bd2 Ng3+ 76. Kd3 Nf5 77. Bg5 Kd7 78. Bf6 Nd4 79. Ke4 Kc6 80. Be7 Nf5 81. Bf6 Nd4 82. Bh4 Ne2 83. Be1 Kd7 84. Bd2 Ng3+ 85. Kd3 Nf5 86. Bc3 Kc7 87. f4 Ng3 88. Bb2 Kd7 89. Bc3 Kc7 90. Be1 Nf5 91. Ke4 Kc6 92. Bc3 Ng3+ 93. Ke3 Nf5+ 94. Kd3 Ng3 95. Be1 Nf5 96. Ke4 Kd7 97. Bc3 Kc6 98. Be1 Kd7 99. Kd3 Kc7 100. Bd2 Kd7 101. Bc3 Kc6 102. Ke4 Ng3+ 103. Ke3 Nf5+ 104. Kd3 Ng3 105. Bb2 Nf5 106. Ke4 Ng3+ 107. Kf3 Nf5 108. Bc3 Kd7 109. Bb2 Kc6 110. Bc3 Kd7 111. Bd2 Kc6 112. Ke4 Ng3+ 113. Kd3 Nf5 114. Ke2 Ng3+ 115. Kf3 Nf5 116. Be1 Nd4+ 117. Ke4 Ne2 118. Ke3 Nd4 119. Bf2 Kd7 120. Kd3 Nf5 121. Ke4 Kc6 122. Kd3 Kd7 123. Bg1 Ng3 124. Be3 Kc6 125. Bf2 Nf5 126. Ke4 Kd7 127. Be1 Nd4 128. Bg3 Ne2 129. Be1 Kc6 130. Ke3 Nd4 131. Bf2 Kd7 132. Kd2 Nf5 133. Kd3 Kc7 134. Be1 Kc6 135. Bd2 Kd7 136. Ke4 Ng3+ 137. Kd3
50-move rule 1/2-1/2

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* " − "]
[*White* "*winglet*"]
[*Black* "*Joli* − 1101"]
[*Result* "1/2 − 1/2"]

[*TimeControl* "300"]
[*Annotator* "2. + 0,002... − 0,27"]

1. c4 c5 2. Nc3 e5 3. e4 Nc6 4. Nf3 Nf6 5. Be2 Bd6 6. O-O Qb6 7. d3 O-O 8. Rb1 Rb8 9. Re1 Re8 10. Be3 Nd4 11. Qd2 Nxe2+ 12. Qxe2 Bf8 13. b4 Qc7 14. Nb5 Qb6 15. Na3 Ng4 16. bxc5 Qa5 17. Nb5 Nxe3 18. Qxe3 f6 19. Qd2 Qxd2 20. Nxd2 Kf7 21. Nb3 a6 22. Nc7 Rd8 23. Rb2 d6 24. Na5 Rd7 25. Nd5 dxc5 26. Nc6 Ra8 27. Na5 Rb8 28. Nc6 Ra8 29. Nb6 bxc6 30. Nxa8 Rxd3 31. Nb6 Bb7 32. Nd5 Bc8 33. Ne3 Ra3 34. Kf1 Ra4 35. Rc2 Be6 36. Rd1 Be7 37. Ke2 h5 38. g3 g6 39. f3 Rb4 40. h4 a5 41. f4 exf4 42. gxf4 f5 43. exf5 gxf5 44. Kd3 Bxh4 45. Rh1 Bg3 46. Ng2 h4 47. Nxh4 Bxf4 48. Rf1 Be5 49. Nxf5 Ke8 50. Ke4 Bb2 51. Rd1 a4 52. Kf4 Kf7 53. Re1 Kf6 54. Nd6 a3 55. Ne4+ Kg7 56. Rg2+ Kf8 57. Nd6 Bd7 58. Kg5 Bc3 59. Rf2+ Kg8 60. Re7 Rb2 61. Rff7 Rg2+ 62. Kf4 Rf2+ 63. Ke4 Rxf7 64. Rxf7 Bh3 65. Kf4 Bb2 66. Ra7 Bg2 67. Nb7 Bf1 68. Ra4 Kf7 69. Nxc5 Ke8 70. Ne4 Kd8 71. Ke3 Kc7 72. Nd2 Bc1 73. c5 Bb5 74. Rb4 Kd7 75. Ke4 Kc8 76. Nc4 Kc7 77. Nd6 Be2 78. Rb7+ Kd8 79. Nf7+ Kc8 80. Nd6+ Kd8 81. Nf7+ Kc8 82. Ra7 Kb8 83. Ra4 Bb5 84. Rb4 Kc7 85. Nd6 Be2 86. Rb6 Bb2 87. Rb7+ Kd8 88. Nf7+ Kc8 89. Nd6+ Kd8 90. Rh7 Bc3 91. Ra7 Bb2 92. Nf7+ Kc8 93. Ne5 Bb5 94. Re7 Bc3 95. Ra7 Bb2 96. Re7 Bc3 97. Rh7 Kb8 98. Rh3 Bb2 99. Rh7 Bc3 100. Rh3 Bb2 101. Rf3 Kc8 102. Rf8+ Kc7 103. Rf7+ Kc8 104. Ra7 Bf1 105. Kf5 Bh3+ 106. Ke4 Bg2+ 107. Nf3 Kd8 108. Ke3 Bh3 109. Ra6 Be6 110. Nd4 Bxa2 111. Nxc6+ Kd7 112. Nd4 Bc4 113. Ra7+ Kc8 114. Nc6 a2 115. Ne7+ Kd8 116. c6 a1=Q 117. c7+ Kxe7 118. Rxa1 Kd7 119. Ra5 Kc8 120. Ke4 Bf6 121. Rc5 Ba6 122. Kd5 Bb2 123. Rc2 Bg7 124. Ke6 Bd3 125. Rc6 Be4 126. Rc4 Bd3 127. Rc6 Be2 128. Rc5 Bd4 129. Rc2 Bd1 130. Rc1 Be2 131. Re1 Bh5 132. Rc1 Be2 133. Ke7 Be3 134. Rc2 Bb5 135. Kd6 Bf4+ 136. Kc5 Bf1 137. Kb6 Be5 138. Rc1 Bg2 139. Rc4 Bd5 140. Rc1 Bf4 141. Rc2 Be4 142. Rc3 Bd5 143. Kc5 Bg2 144. Rc2 Bf3 145. Rc3 Bg2 146. Rc2 Bf3 147. Kb6 Be4 148. Rc4 Be3+ 149. Ka5 Bd5 150. Rc2 Bb3 151. Rc6 Bd5 152. Rc2 Bd4 153. Kb5 Be4 154. Rc1 Bb7 155. Rc4 Be5 156. Kb6 Bd5 157. Rc1 Bb7 158. Rc2 Bd4+ 159. Ka5 Bd5 160. Kb5 Be4 161. Rc1 Bf6 162. Ka5 Be5 163. Kb6 Bf4 164. Rc4 Be3+ 165. Ka5 Bd3 166. Rc6 Bd4 167. Rc1 Be4 168. Kb5
repetition draw 1/2-1/2

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]

[*Date* "2013,11,07"]
[*Round* "−"]
[*White* "*Joli* − 1101"]
[*Black* "*winglet*"]
[*Result* "1 − 0"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,012... − 0,35"]

1. c4 c5 2. Nf3 Nf6 3. Nc3 Nc6 4. e3 d5 5. cxd5 Nxd5 6. Bb5 Qd6 7. O-O Bf5 8. Nh4 Bd7 9. Ne4 Qh6 10. Nf3 Nc7 11. Be2 e5 12. d4 cxd4 13. exd4 Qg6 14. Nfg5 h5 15. dxe5 Nxe5 16. Bf4 Ne6 17. Nxe6 Qxe6 18. Bxh5 O-O-O 19. Ng5 Qf5 20. Bxe5 Qxe5 21. Rc1+ Bc5 22. Nxf7 Qxh5 23. Nxh8 Rxh8 24. Qxh5 Rxh5 25. b4 b6 26. bxc5 bxc5 27. Rfe1 g6 28. Rcd1 Rg5 29. f4 Rf5 30. Re7 Bb5 31. g3 a6 32. Rd6 g5 33. Rg6 Bc4 34. Rgg7 Rd5 35. Rg8+ Rd8 36. Rxg5 Rd1+ 37. Kf2 Rd2+ 38. Kf3 Rxh2 39. Rxc5+ Kd8 40. Ra7 Rc2 41. Ke4 Re2+ 42. Kd4 Bb5 43. a4 Bxa4 44. Rxa6 Bb3 45. Rb5 Be6 46. Re5 Bc8 47. Ra7 Rd2+ 48. Ke3 Rd7 49. Ra8 Rc7 50. Rh5 Kd7 51. Rh6 Bb7 52. Ra7 Rc3+ 53. Kd4 Rb3 54. Rh7+ Kc6 55. f5 Rb4+ 56. Kc3 Rb5 57. Rhxb7 Rxf5 58. Kd4 Rf2 59. Rc7+ Kd6 60. Rd7+ Ke6 61. Re7+ Kd6 62. Rad7+ Kc6 63. Rc7+ Kd6 64. Red7+ Ke6 65. Re7+ Kd6 66. Rcd7+ Kc6 67. Rb7 Kd6 68. Red7+ Ke6 69. Re7+ Kd6 70. Rec7 Rd2+ 71. Ke4 Re2+ 72. Kf3 Rh2 73. Kf4 Rf2+ 74. Ke3 Rg2 75. Kf4 Rf2+ 76. Ke3 Ra2 77. Ra7 Rh2 78. Kd4 Rd2+ 79. Ke4 Re2+ 80. Kf3 Rh2 81. Kf4 Rf2+ 82. Ke3 Rb2 83. Kf4 Ke6 84. Re7+ Kd5 85. Rad7+ Kc4 86. Rd1 Rf2+ 87. Ke3 Rg2 88. Rc7+ Kb3 89. Kf3 Rb2 90. Rd8 Ka2 91. g4 Rb4 92. Rd1 Rb2 93. g5 Rb3+ 94. Kf4 Rb4+ 95. Ke5 Rb5+ 96. Kf4 Rb4+ 97. Kf3 Rb3+ 98. Ke4 Rb4+ 99. Kf5 Rb5+ 100. Kf6 Rb2 101. Rg1 Rf2+ 102. Ke5 Kb2 103. g6 Re2+ 104. Kf4 Rf2+ 105. Ke3 Rf8 106. g7 Re8+ 107. Kf4 Rg8 108. Rg6 Rb8 109. g8=Q Rb4+ 110. Ke3 Ka3 111. Ra6+ Ra4 112. Rb7 Rxa6 113. Qb3#
Checkmate 1-0

A.2. Joli1001 vs Winglet

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]

[*Round* “ − ”]
[*White* “*Joli* − 1001”]
[*Black* “*winglet*”]
[*Result* “0 − 1”]
[*TimeControl* “300”]
[*Annotator* “2. + 0,002... − 0,35”]

1. c4 c5 2. Nc3 e5 3. e4 Nf6 4. Nf3 Nc6 5. Bd3 Be7 6. O-O O-O 7. Nd5 d6 8. Nxe7+ Qxe7 9. Re1 Bg4 10. h3 Bxf3 11. Qxf3 Rae8 12. b3 Nb4 13. Bb1 Nc6 14. d3 Nd4 15. Qg3 Nh5 16. Qg4 Nf6 17. Qg3 Qe6 18. Qg5 b6 19. Be3 Nc6 20. a3 h6 21. Qg3 Rd8 22. Bc2 Nd4 23. Bd1 Kh8 24. b4 a5 25. bxa5 bxa5 26. Bd2 Ra8 27. Ba4 Rfd8 28. Rab1 Nh5 29. Qe3 Nf4 30. Rb7 Qg6 31. g4 Qf6 32. Bd1 Nde6 33. Rb5 Ng5 34. Bxa5 Ngxh3+ 35. Kf1 Rd7 36. Bc2 Rda7 37. Bc3 Qh4 38. Qg3 Qxg3 39. fxg3 Rxa3 40. Bxe5 Nxd3 41. Bxd3 dxe5 42. Rd1 Rd8 43. Bc2 Rxd1+ 44. Bxd1 Ng5 45. Rxc5 f6 46. Rb5 Rxg3 47. Rb3 Rxb3 48. Bxb3 Nxe4 49. Ke2 Kg8 50. c5+ Kf8 51. c6 Ke7 52. Ke3 Nc5 53. Bd5 Kd6 54. Bg2 Ne6 55. Be4 Nd4 56. Bd5 Nxc6 57. Ke4 g6 58. Bf7 Ne7 59. Ke3 f5 60. gxf5 gxf5 61. Bb3 Nd5+ 62. Kd3 Kc5 63. Kd2 Kd4 64. Ba4 e4 65. Bc6 Nc3 66. Kc2 e3 67. Bf3 f4 68. Bh5 Ne4 69. Kd1 Nc3+ 70. Kc2 Ne4 71. Kd1 Nd2 72. Ke2 Nb1 73. Kd1 Nd2 74. Ke2 Nc4 75. Kf3 Ke5 76. Be8 Nd2+ 77. Ke2 Nb3 78. Kd3 Nc5+ 79. Ke2 Ne6 80. Bh5 Nc5 81. Bg6 Nb3 82. Kd3 Nc5+ 83. Ke2 Nb3 84. Kd3 Nc1+ 85. Kc2 e2 86. Kd2 f3 87. Ke1 Nb3 88. Bc2 Kf4 89. Kf2 Nd4 90. Bb1 h5 91. Bd3 h4 92. Ke1 Ke3 93. Bxe2 Nxe2 94. Kd1 f2 95. Kc2 f1=Q 96. Kb3 Nd4+ 97. Kc3 Qd3+ 98. Kb4 Qb5+ 99. Kc3 Qb3#
Checkmate 0-1

[*Event* “*ComputerChessGame*”]
[*Site* “*ALICE*”]
[*Date* “2013,11,07”]
[*Round* “ − ”]
[*White* “*winglet*”]
[*Black* “*Joli* − 1001”]
[*Result* “1/2 − 1/2”]
[*TimeControl* “300”]
[*Annotator* “2. + 0,002... − 0,36”]

1. c4 c5 2. Nc3 e5 3. e4 Nc6 4. Nf3 Nf6 5. Be2 Be7 6. O-O O-O 7. d3 d6 8.

Be3 Nd4 9. Rb1 Qc7 10. Re1 Nxe2+ 11. Qxe2 Bd7 12. Rbd1 Rad8 13. a3 a5 14. Qd2 Rfe8 15. Qc2 Qc6 16. Qe2 a4 17. Qc2 Ra8 18. h3 Ra6 19. Qd2 Qc7 20. g4 h6 21. Qc2 Qb6 22. Kh1 Kh7 23. Kg1 Kg8 24. Ne2 Qc6 25. Nc3 Rb6 26. Nd2 Ra6 27. Nf3 Qc7 28. Qd2 Ra7 29. Qc2 Qb6 30. d4 cxd4 31. Nxd4 exd4 32. Bxd4 Qa5 33. Bxa7 Qxa7 34. Nd5 Nxd5 35. exd5 b5 36. cxb5 Bxb5 37. Qc3 Qb6 38. Qb4 Qb7 39. Re4 Rb8 40. Rde1 Bf6 41. b3 axb3 42. Re8+ Rxe8 43. Rxe8+ Kh7 44. Qxb3 Qa6 45. Rf8 Kg6 46. Qc2+ Bd3 47. Qc7 Qxa3 48. Qxf7+ Kg5 49. Qh5+ Kf4 50. g5 hxg5 51. Qg4+ Ke5 52. Qe6+ Kd4 53. Qe3+ Kxd5 54. Rc8 Qa1+ 55. Rc1 Qd4 56. Qg3 Bc2 57. Qf3+ Be4 58. Qb3+ Ke5 59. Rd1 Qc5 60. Qg3+ Ke6 61. Qg4+ Ke5 62. Qh5 d5 63. Qe8+ Be7 64. Qf7 g6 65. Qg7+ Bf6 66. Qd7 d4 67. Re1 Qb4 68. Qc7+ Kf5 69. Qd7+ Ke5 70. Qc7+ Kd5 71. Qf7+ Ke5 72. Qc7+
repetition draw 1/2-1/2

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* "−"]
[*White* "*Joli* − 1001"]
[*Black* "*winglet*"]
[*Result* "1 − 0"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,002... − 0,27"]

1. d4 d5 2. Nc3 Nc6 3. e4 e6 4. e5 Bb4 5. Qg4 Bxc3+ 6. bxc3 Kf8 7. Nf3 Nge7 8. Bd3 Kg8 9. Bg5 Bd7 10. O-O Qf8 11. Rfe1 Re8 12. Rab1 b6 13. c4 Rd8 14. Qg3 Re8 15. Qg4 Rd8 16. Qg3 Re8 17. Bd2 Rd8 18. Rbd1 Qe8 19. a3 h6 20. Qg4 a5 21. Qg3 Rc8 22. Qf4 a4 23. cxd5 exd5 24. Bc3 Be6 25. Bb5 Ra8 26. Bd3 Qd7 27. Nh4 Bg4 28. f3 Be6 29. Qg3 Na7 30. Bb4 Re8 31. f4 Nf5 32. Nxf5 Bxf5 33. c4 Bxd3 34. Qxd3 Nc6 35. Bc3 Ne7 36. Bb4 dxc4 37. Qxc4 b5 38. Qc5 Nd5 39. Bd2 Rh7 40. f5 h5 41. Rb1 c6 42. Rbc1 Ne7 43. e6 Qd5 44. Qxd5 cxd5 45. Bb4 Nxf5 46. exf7+ Kxf7 47. Rc7+ Ne7 48. Rexe7+ Rxe7 49. Rxe7+ Kf6 50. Re5 g6 51. Rxd5 Rb7 52. Kf2 Ke6 53. Re5+ Kf7 54. Ke3 Kg7 55. h4 Kg8 56. d5 Kf7 57. d6 Kf6 58. Bc3 Kf7 59. Ke4 Rd7 60. Kd5 Ra7 61. Bd4 Rd7 62. Kc6 Rd8 63. Kc7 Rg8 64. d7 Rf8 65. d8=Q Rxd8 66. Kxd8 Kf8 67. Rxb5 Kf7 68. Kd7 Kg8 69. Ke6 Kh7 70. Rb7+ Kg8 71. Rb8+ Kh7 72. Rh8#
Checkmate 1-0

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* "−"]
[*White* "*winglet*"]
[*Black* "*Joli* − 1001"]
[*Result* "1 − 0"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,012... − 0,26"]

1. d4 d5 2. Nc3 Nc6 3. e4 e6 4. Bb5 dxe4 5. Bxc6+ bxc6 6. Nxe4 Qd5 7. Qf3 Ne7 8. Be3 Rb8 9. b3 Nf5 10. Ne2 Nxe3 11. Qxe3 c5 12. O-O Bb7 13. f3 cxd4 14. Nxd4 Qe5 15. Rfe1 c5 16. Ne2 f5 17. Qc3 Qxc3 18. N4xc3 Be7 19. Na4 Kf7 20. Rad1 Rhd8 21. Rxd8 Rxd8 22. Kf2 e5 23. c4 Bc6 24. Nec3 Ke6 25. Ke2 e4 26. fxe4 fxe4 27. Rd1 Rxd1 28. Kxd1 g5 29. Ke2 Ke5 30. Ke3 g4 31. Ke2 h5 32. Kf2 Kf5 33. a3 Bf6 34. b4 cxb4 35. axb4 Bd4+ 36. Ke2 a6 37. h4 Kf4 38. b5 axb5 39. cxb5 Bb7 40. Kf1 Bxc3 41. Nxc3 Kg3 42. Ke2 e3 43. Kxe3 Bxg2 44. Ne4+ Kxh4 45. b6 Bxe4 46. Kxe4 g3 47. Kf3 Kh3 48. b7 g2 49. Kf2 h4 50. b8=Q Kg4 51. Kxg2 h3+ 52. Kf2 Kf5 53. Ke3 Ke6 54. Qh2 Kd5 55. Qxh3 Kd6 56. Kd4 Kc6 57. Qe6+ Kb5 58. Qd6 Ka4 59. Qb6 Ka3 60. Kc3 Ka4 61. Qb4#
Checkmate 1-0

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* "−"]
[*White* "*Joli* − 1001"]
[*Black* "*winglet*"]
[*Result* "1 − 0"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,052... − 0,35"]

1. e4 e5 2. Nf3 Nf6 3. Nc3 Bb4 4. Nxe5 Qe7 5. Ng4 Nxe4 6. Nd5 Qd6 7. Nxb4 Qxb4 8. Be2 d5 9. O-O Bxg4 10. Bxg4 O-O 11. c3 Qd6 12. d4 Nc6 13. g3 f5 14. Bf3 Rad8 15. Bf4 Qd7 16. Qb3 Na5 17. Qb4 b6 18. Rad1 Nc4 19. Rfe1 Qc6 20. a4 Rfe8 21. Bh5 Re6 22. h4 Nf6 23. Rxe6 Qxe6 24. Bf3 c6 25.

Qb3 Ne4 26. Qc2 Ncd6 27. b3 Kf7 28. Be5 Kg8 29. Bf4 Kf7 30. Be5 Kg8 31. Qd3 Nf7 32. Bc7 Rd7 33. Bf4 Rd8 34. Re1 Nfd6 35. h5 Qd7 36. Bg2 a5 37. f3 Nf6 38. h6 Nh5 39. Bg5 Re8 40. Rxe8+ Qxe8 41. Kh2 gxh6 42. Bh4 Qe1 43. Kh3 Ng7 44. Qa6 Nde8 45. c4 Qb4 46. c5 bxc5 47. Qxc6 Qxd4 48. f4 Kh8 49. Bxd5 Qe3 50. Qxh6 Qd4 51. Qc6 Qe3 52. Bd8 Nh5 53. Bf3 Nhg7 54. Bxa5 Qxb3 55. Qxc5 Qf7 56. Bc3 Nc7 57. Kg2 Kg8 58. Qe5 Qa2+ 59. Kf1 Qb1+ 60. Ke2 Qc2+ 61. Ke3 Qc1+ 62. Kf2 Qc2+ 63. Ke3 Qc1+ 64. Kd3 Qf1+ 65. Be2 Qb1+ 66. Kd2 Qa2+ 67. Ke3 Nce6 68. a5 Qb1 69. Bc4 Qc1+ 70. Kf3 Qh1+ 71. Kf2 Qh2+ 72. Kf1 Qh1+ 73. Kf2 Qh2+ 74. Kf3 Qh1+ 75. Ke3 Qc1+ 76. Ke2 Qc2+ 77. Kf3 Qe4+ 78. Qxe4 fxe4+ 79. Kxe4 Kf8 80. a6 Nc7 81. a7 Nh5 82. g4 Ng7 83. Bb4+ Ke8 84. Bd6 Na8 85. Bd5 Nb6 86. Bc5 Na8 87. Bxa8 Ne6 88. Bc6+ Kf7 89. Bd5 Kf6 90. Bxe6 Kxe6 91. a8=Q Kd7 92. Qb7+ Kd8 93. Qe7+ Kc8 94. Bd6 h6 95. Qc7#
Checkmate 1-0

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* "−"]
[*White* "*winglet*"]
[*Black* "*Joli* − 1001"]
[*Result* "0 − 1"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,052... − 0,37"]

1. e4 e5 2. Nf3 Nf6 3. Nc3 Nc6 4. Bc4 Nxe4 5. Nxe4 d5 6. Bd3 f5 7. Bb5 fxe4 8. Nxe5 Qf6 9. d4 Be7 10. Bxc6+ bxc6 11. O-O c5 12. f3 cxd4 13. Qxd4 Qb6 14. Be3 Qxd4 15. Bxd4 c5 16. Bf2 Bf6 17. Bg3 e3 18. Rfe1 d4 19. Nc4 Be6 20. Nd6+ Kf8 21. Ne4 Be7 22. c3 Rd8 23. Be5 Rd5 24. Bb8 a6 25. Ba7 Kf7 26. b4 cxb4 27. Bxd4 bxc3 28. Bxe3 Rc8 29. Re2 c2 30. Rf2 Ra5 31. a4 Bd7 32. Rc1 Bxa4 33. Ra1 Bb5 34. Rc1 Ra2 35. Rd2 Rc6 36. Rd7 Ke6 37. Rb7 Ba3 38. Re1 c1=Q 39. Bxc1 Rxc1 40. Rxc1 Bxc1 41. h3 Bf4 42. g4 Be3+ 43. Kh1 Bf1 44. Rb3 Bg2+ 45. Kh2 Bf4+ 46. Kg1 Bxh3 47. Rd3 Rg2+ 48. Kh1 Rc2 49. Kg1 h6 50. Nf2 Rc1+ 51. Rd1 Rxd1+ 52. Nxd1 Ke5 53. Kf2 Kd4 54. Ke2 a5 55. Nf2 Bg2 56. Ne4 g5 57. Nf6 a4 58. Kd1 Be5 59. Ng8 Bxf3+ 60. Kc2 Bg7 61. Ne7 Bxg4 62. Kd2 Bd7 63. Kc1 a3 64. Kb1 Be6 65. Nf5+ Bxf5+ 66. Ka2 Bf8 67. Kb3 Kd3 68. Ka4 Kc4 69. Ka5 Bc5 70. Ka6 Be4 71. Ka5 a2 72. Ka4 a1=Q#

Checkmate 0-1

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* "−"]
[*White* "*Joli* − 1001"]
[*Black* "*winglet*"]
[*Result* "1 − 0"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,002... − 0,26"]

1. Nf3 Nf6 2. d4 d5 3. Nc3 e6 4. e3 Bd6 5. Bb5+ Bd7 6. Ne5 Bxb5 7. Nxb5 Bxe5 8. dxe5 Ne4 9. O-O Nc6 10. Qh5 O-O 11. Nd4 Qd7 12. f3 Nc5 13. Bd2 a5 14. Rad1 b6 15. Rfe1 Rad8 16. Qg5 Rfe8 17. b3 Ne7 18. Bc3 c6 19. Ne2 Qc7 20. Ng3 h6 21. Qg4 b5 22. e4 Nd7 23. f4 Qb6+ 24. Bd4 c5 25. Bb2 d4 26. Nh5 g6 27. Ng3 Nc6 28. c3 b4 29. cxd4 cxd4 30. Qh4 d3+ 31. Kf1 Kg7 32. Nh1 Nd4 33. Bxd4 Qxd4 34. Nf2 Nc5 35. Ng4 Rh8 36. Qf6+ Kg8 37. Qh4 Kg7 38. Qf6+ Kg8 39. Qe7 Na6 40. h4 Nc5 41. h5 gxh5 42. Nf6+ Kg7 43. Nxh5+ Kg6 44. g4 Rhg8 45. Qf6+ Kh7 46. Qxf7+ Kh8 47. Nf6 Qd7 48. Nxd7 Rxd7 49. Qf6+ Kh7 50. Qh4 Rh8 51. Kf2 Rg8 52. Ke3 Rgg7 53. Rh1 Kg8 54. Rc1 d2 55. Rxc5 d1=N+ 56. Kf3 Rg6 57. Rc7 Rxc7 58. Qd8+ Kh7 59. Qxc7+ Rg7 60. Qd8 Rxg4 61. Rxd1 Rg7 62. Qxa5 Rb7 63. Qa6 Re7 64. Qd6 Ra7 65. Qxe6 Rg7 66. Qf5+ Kh8 67. Rd8+ Rg8 68. Qf6+ Kh7 69. Rd7+ Rg7 70. Qxg7#
Checkmate 1-0

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* "−"]
[*White* "*winglet*"]
[*Black* "*Joli* − 1001"]
[*Result* "0 − 1"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,002... − 0,26"]

1. Nf3 Nf6 2. d4 d5 3. Nc3 e6 4. Bg5 h6 5. Bxf6 Qxf6 6. a3 Nc6 7. e4

dxe4 8. Nxe4 Qg6 9. Qd3 f5 10. Qb5 a6 11. Ne5 Qh7 12. Qa4 fxe4 13. Bb5 Bd7 14. Bxc6 bxc6 15. Qb3 c5 16. Qb7 Rd8 17. Nc6 Rc8 18. Na7 Rd8 19. Nc6 e3 20. O-O-O e2 21. Rde1 Rc8 22. Na7 Rd8 23. Nc6 Qf5 24. Nxd8 Kxd8 25. Rxe2 cxd4 26. Rd2 Bd6 27. Rxd4 Qxf2 28. Rhd1 Rf8 29. Qa8+ Bc8 30. Qc6 Ke7 31. g3 Kf7 32. R4d3 Kg8 33. R3d2 Qf7 34. c4 Qf5 35. Re1 Rd8 36. Rc2 Qg5+ 37. Kb1 Qg6 38. c5 Bd7 39. Qe4 Qxe4 40. Rxe4 Be7 41. c6 Be8 42. Rxe6 Rd1+ 43. Rc1 Rxc1+ 44. Kxc1 Kf8 45. Kc2 Bd6 46. Re1 Bxc6 47. Kd3 Kf7 48. Kd4 Bb7 49. Rf1+ Kg6 50. Rd1 a5 51. Rd2 a4 52. Kc4 Bc8 53. Kd4 Kf6 54. Rf2+ Ke7 55. Re2+ Be6 56. Ke4 g5 57. Kd3 Kd7 58. Re1 g4 59. Ke4 c6 60. Re3 c5 61. Re1 Kc6 62. Rd1 c4 63. Kd4 Bd5 64. Rd2 Bc7 65. Re2 Kd6 66. Kc3 Bf3 67. Re8 Kd5 68. Ra8 Bd1 69. Kd2 Bb3 70. Rg8 Be5 71. Rxg4 Bxb2 72. Rh4 Bf6 73. Rh5+ Bg5+ 74. Rxg5+ hxg5 75. Kc3 Ke4 76. Kd2 Kf3 77. h4 gxh4 78. gxh4 Kg4 79. Kc3 Kxh4 80. Kd4 Kg4 81. Kc3 Kf5 82. Kd4 Kf4 83. Kc3 Ke3 84. Kb4 Kd4 85. Kb5 c3 86. Kc6 c2 87. Kd6 c1=Q 88. Ke7 Qc6 89. Kd8 Qb7 90. Ke8 Be6 91. Kd8 Qd7#
Checkmate 0-1

A.3. Joli1101 vs Joli1001

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* "−"]
[*White* "*Joli* − 1101"]
[*Black* "*Joli* − 1001"]
[*Result* "1 − 0"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,092... − 0,37"]

1. e4 e5 2. Nc3 Nc6 3. Nf3 Nf6 4. Bb5 Bc5 5. O-O Qe7 6. d3 a6 7. Bc4 O-O 8. Bg5 b5 9. Bb3 Qd6 10. Nd5 Nxd5 11. Bxd5 Ra7 12. c3 h6 13. Bh4 Ne7 14. Bxe7 Qxe7 15. Nxe5 Qxe5 16. d4 Qe7 17. dxc5 c6 18. Bb3 Qxc5 19. Qd4 Qxd4 20. cxd4 Bb7 21. d5 cxd5 22. exd5 d6 23. Rfe1 Rd8 24. Re7 a5 25. Rc1 a4 26. Bd1 Rb8 27. Rcc7 b4 28. Rxf7 g5 29. Rh7 b3 30. a3 Ra5 31. Rcg7+ Kf8 32. Rf7+ Kg8 33. Rhg7+ Kh8 34. Rh7+ Kg8 35. Rhg7+ Kh8 36. Rxb7 Rxb7 37. Rxb7 Rxd5 38. Bg4 Rd2 39. Kf1 Rxb2 40. Bd7 Rb1+ 41.

Ke2 Rb2+ 42. Ke3 Ra2 43. Bxa4 b2 44. Bc2 Rxa3+ 45. Kd4 Ra1 46. Rxb2 Rf1 47. Bb3 Rh1 48. h3 Rg1 49. f3 Kg7 50. Kd5 Rh1 51. Kxd6 Kf6 52. Bd5 Rg1 53. Re2 Kg6 54. Ke5 Rf1 55. h4 gxh4 56. Re4 Rh1 57. Rg4+ Kh7 58. Be4+ Kh8 59. Kf6 Ra1 60. Kf7 Ra7+ 61. Kf8 Ra8+ 62. Bxa8 h5 63. Rg7 h3 64. Be4 hxg2 65. Rh7#
Checkmate 1-0

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* " − "]
[*White* "*Joli* − 1001"]
[*Black* "*Joli* − 1101"]
[*Result* "1/2 − 1/2"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,052... − 0,39"]

1. e4 e5 2. Nf3 Nf6 3. Nc3 Nc6 4. Bb5 Nd4 5. Bc4 Nxe4 6. Nxd4 exd4 7. Qe2 dxc3 8. Qxe4+ Be7 9. dxc3 c6 10. Bd3 d5 11. Qf3 O-O 12. Be3 Bd6 13. O-O Qh4 14. h3 Be6 15. Rad1 Rfe8 16. Rfe1 b6 17. Bd4 Qg5 18. c4 h6 19. Bc3 Rad8 20. Bd2 Qh4 21. cxd5 Bxd5 22. Rxe8+ Rxe8 23. Qf5 Be5 24. Qh7+ Kf8 25. Re1 Qf6 26. Bb4+ c5 27. Bb5 g5 28. Bc3 Re6 29. Rxe5 Rxe5 30. Bxe5 Qxe5 31. Qxh6+ Ke7 32. c4 Qe1+ 33. Kh2 Qe5+ 34. g3 Be6 35. b3 Qf5 36. Kg2 Qe4+ 37. Kh2 Qf5 38. Kg1 Qb1+ 39. Kg2 Qe4+ 40. Kg1 Qe1+ 41. Kg2 Qe4+
repetition draw 1/2-1/2

[*Event* "*ComputerChessGame*"]
[*Site* "*LEELOO*"]
[*Date* "2013,11,07"]
[*Round* " − "]
[*White* "*Joli* − 1101"]
[*Black* "*Joli* − 1001"]
[*Result* "1 − 0"]
[*TimeControl* "7200"]
[*Annotator* "2. + 0,022... − 0,26"]

1. d4 d5 2. Nc3 Nc6 3. e4 e6 4. e5 Bd7 5. Nf3 Bb4 6. Bd3 Nge7 7. O-O

O-O 8. Bd2 Ba5 9. Be3 Bb6 10. a3 Be8 11. g4 Nxd4 12. Nxd4 c5 13. Ndb5 d4 14. Bg5 dxc3 15. Nxc3 h6 16. Bh4 Qc7 17. Qe2 Ng6 18. Bg3 Rd8 19. Rad1 Bc6 20. Nb5 Qe7 21. Be4 Bxe4 22. Qxe4 Rd7 23. Rfe1 Rfd8 24. Rxd7 Qxd7 25. Nd6 Rb8 26. b3 Ne7 27. Rd1 Nc6 28. b4 cxb4 29. axb4 a6 30. c3 Rd8 31. Bh4 Rb8 32. Bg3 Rd8 33. Re1 Qe7 34. Nc4 Ba7 35. Nd6 Bb6 36. Bf4 Qd7 37. Kg2 Qe7 38. Kg1 Qc7 39. Rc1 Qd7 40. Re1 Ba7 41. Rd1 Qc7 42. Rb1 Bb6 43. b5 axb5 44. Rxb5 Rb8 45. Nc4 Ba7 46. Nd6 Bb6 47. Qb1 Ba5 48. Qd3 Rd8 49. Qc4 b6 50. Rb1 Qd7 51. Qe4 Bxc3 52. Rxb6 Nd4 53. Ra6 Qc7 54. Ra8 Rxa8 55. Qxa8+ Kh7 56. Kg2 Qc6+ 57. Qxc6 Nxc6 58. Nxf7 g5 59. Bxg5 hxg5 60. Nxg5+ Kg6 61. Nxe6 Bxe5 62. f4 Kf6 63. f5 Bd6 64. h4 Ne5 65. Kh3 Nf3 66. Nd8 Nd4 67. Nb7 Bb4 68. h5 Ke5 69. h6 Nf3 70. Kg3 Ng5 71. Kh4 Be7 72. Kh5 Nh3 73. Kg6 Nf4+ 74. Kf7 Nd5 75. h7 Bf6 76. Nc5 Bh8 77. Kg8 Bf6 78. Kf7 Bh8 79. Kg8 Bf6 80. Ne6 Ne7+ 81. Kf8 Nxf5 82. gxf5 Bh8 83. Kg8 Kf6 84. Kxh8 Kf7 85. Nd8+ Kf8 86. f6 Ke8 87. Kg7 Kxd8 88. h8=Q+ Kd7 89. Qh1 Kd6 90. Qe4 Kc7 91. f7 Kd7 92. f8=Q Kc7 93. Qeb4 Kc6 94. Qfd6#
Checkmate 1-0

[*Event* "*ComputerChessGame*"]
[*Site* "*LEELOO*"]
[*Date* "2013,11,07"]
[*Round* " − "]
[*White* "*Joli* − 1001"]
[*Black* "*Joli* − 1101"]
[*Result* "0 − 1"]
[*TimeControl* "7200"]
[*Annotator* "2. + 0,002... − 0,28"]

1. d4 d5 2. Nc3 Nc6 3. e4 e6 4. e5 Bd7 5. Nf3 Bb4 6. Bd3 Nge7 7. O-O O-O 8. Be3 Rb8 9. Qe2 Ng6 10. Rad1 Bxc3 11. bxc3 Qe7 12. Qd2 Qa3 13. Ra1 Na5 14. Qc1 Qxc1 15. Bxc1 Rfe8 16. Rb1 Nc4 17. Re1 b5 18. h4 Rb6 19. h5 Ne7 20. h6 gxh6 21. g4 Ra6 22. Ra1 Bc6 23. Rf1 Ra4 24. Re1 Ng6 25. Re2 Rb8 26. Bxh6 Rb6 27. Bg5 Rba6 28. Ree1 Bd7 29. Kf1 Nb2 30. Be2 Rxa2 31. Rxa2 Rxa2 32. Rc1 Na4 33. Bd2 Ra3 34. c4 dxc4 35. Bb4 Ra2 36. Ke1 Nf4 37. Bd2 Nxe2 38. Kxe2 c5 39. dxc5 Nxc5 40. Ke3 c3 41. Be1 b4 42. Kd4 Na6 43. g5 Bc6 44. Ke3 Ra5 45. Rd1 Bxf3 46. Kxf3 Rxe5 47. Kf4 Re2 48. Rc1 Nc5 49. Kf3 Re5 50. Kg4 Re4+ 51. Kh5 Re2 52. f4 b3 53. Rd1 Rxc2 54. Kg4 b2 55. Rd8+ Kg7 56. Rd1 Rd2 57. Bxd2 c2 58. Bc3+ Kg6 59. Rh1

f5+ 60. Kf3 c1=Q 61. Rh6+ Kf7 62. Rxh7+ Ke8 63. Rh8+ Kd7 64. Rh7+ Kc6 65. Bxb2 Qxb2 66. Re7 Qd4 67. Rxa7 Ne4 68. Ra6+ Kd7 69. Ra2 Qg1 70. g6 Qg3+ 71. Ke2 Nc3+ 72. Kd2 Nxa2 73. g7 Qxg7 74. Kd3 Qc3+ 75. Ke2 Nc1+ 76. Kf2 Qd2+ 77. Kg3 Qe2 78. Kh3 Nd3 79. Kg3 Qf2+ 80. Kh3 Nxf4#
Checkmate 0-1

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* " − "]
[*White* "*Joli* − 1101"]
[*Black* "*Joli* − 1001"]
[*Result* "1 − 0"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,012... − 0,35"]

1. c4 c5 2. Nf3 e6 3. Nc3 d5 4. d4 cxd4 5. Nxd4 Nf6 6. cxd5 Nxd5 7. Bd2 Be7 8. e4 Nxc3 9. Bxc3 O-O 10. Be2 Bf6 11. O-O e5 12. Nf5 Nc6 13. Bc4 Bxf5 14. exf5 Nd4 15. Qg4 Qd7 16. Qe4 Rac8 17. Bd3 Nb5 18. Bb4 Nd6 19. Qd5 Rfd8 20. Rad1 Qc7 21. Qb3 a5 22. Bc3 Qc5 23. Rfe1 Ra8 24. Bb1 a4 25. Qd5 Qxd5 26. Rxd5 Nc4 27. Be4 b6 28. b3 axb3 29. axb3 Ra3 30. Rxd8+ Bxd8 31. Bd5 b5 32. Bxc4 bxc4 33. Rxe5 f6 34. Re8+ Kf7 35. Rxd8 Rxb3 36. Be1 Rb2 37. Rd4 Rc2 38. Rd7+ Kf8 39. Bb4+ Ke8 40. Re7+ Kd8 41. Re1 Rb2 42. Be7+ Kd7 43. Bf8 g5 44. f4 c3 45. fxg5 c2 46. Ba3 Rb1 47. Kf2 fxg5 48. Bc1 g4 49. Kg3 Rb4 50. f6 h5 51. f7 h4+ 52. Kxh4 g3+ 53. Kxg3 Rb8 54. Ba3 Rb3+ 55. Kf2 Rb8 56. Re7+ Kc6 57. Re8 Kb7 58. f8=Q Rxe8 59. Qxe8 Kc7 60. Qe4 Kd7 61. Qxc2 Ke6 62. Qc6+ Kf7 63. Qh6 Ke8 64. Qg7 Kd8 65. Bd6 Ke8 66. Qe7#
Checkmate 1-0

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* " − "]
[*White* "*Joli* − 1001"]
[*Black* "*Joli* − 1101"]
[*Result* "1 − 0"]

[*TimeControl* "300"]
[*Annotator* "2. + 0,002... − 0,27"]

1. c4 c5 2. Nc3 e5 3. e4 Nc6 4. Nf3 Nf6 5. Bd3 Be7 6. O-O O-O 7. Nd5 Nxd5 8. cxd5 Nb4 9. Be2 d6 10. d3 Qb6 11. Qb3 Bd7 12. Re1 a5 13. Be3 a4 14. Qc4 a3 15. b3 Rfe8 16. Nd2 Qa6 17. Qc3 Rad8 18. Nf3 Bb5 19. Red1 f5 20. Bf1 Nxd3 21. exf5 e4 22. Bxd3 Bxd3 23. Bg5 Be2 24. Bxe7 Rxe7 25. f6 gxf6 26. Qxf6 Rde8 27. Nh4 Rf7 28. Qg5+ Rg7 29. Qe3 Bxd1 30. Rxd1 Re5 31. g3 Rgg5 32. Ng2 Rxd5 33. Rxd5 Rxd5 34. Qxe4 Rd1+ 35. Ne1 Ra1 36. Qd5+ Kf8 37. Qf5+ Kg7 38. Qg5+ Kf7 39. Qf5+ Kg8 40. Qg5+ Kf7 41. Qf5+ Ke7 42. Qxh7+ Ke6 43. Qg8+ Ke5 44. Qg7+ Kd5 45. Qxa1 Qe2 46. Qb1 b5 47. Qf5+ Kc6 48. Nc2 Kb6 49. Kg2 Ka5 50. g4 d5 51. g5 Qd2 52. g6 Qh6 53. f4 b4 54. Ne3 Qg7 55. Qf7 Qb2+ 56. Kf3 Qxh2 57. Qh7 Qg1 58. g7 d4 59. Nc4+ Kb5 60. g8=Q Qf1+ 61. Ke4 Qe2+ 62. Kd5 Qf3+ 63. Ke5 Qe2+ 64. Kd6 Qe5+ 65. fxe5 Ka6 66. Qc7 d3 67. Qb6#
Checkmate 1-0

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* " − "]
[*White* "*Joli* − 1001"]
[*Black* "*Joli* − 1101"]
[*Result* "0 − 1"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,002... − 0,28"]

1. Nf3 Nf6 2. d4 d5 3. Nc3 e6 4. e3 Bd6 5. Bb5+ Nbd7 6. O-O O-O 7. Rb1 c6 8. Be2 e5 9. dxe5 Nxe5 10. Nxe5 Bxe5 11. Bd2 Qe7 12. Re1 Bf5 13. a3 Rad8 14. g4 Bg6 15. h3 Ne4 16. Nxe4 dxe4 17. c3 Bb8 18. Kf1 Qf6 19. Bc4 Qd6 20. h4 Qh2 21. Ke2 Qh3 22. Rg1 Bh2 23. h5 Qf3+ 24. Kf1 Bxg1 25. Qxf3 exf3 26. hxg6 hxg6 27. Rd1 Rd7 28. Kxg1 Rfd8 29. Rb1 Rxd2 30. Bb3 Kf8 31. a4 Ke7 32. a5 f5 33. g5 Rh8 34. Bd1 Rh3 35. b3 Kd6 36. c4 c5 37. Ra1 Ke5 38. Rc1 f4 39. e4 Ra2 40. Rc3 Ra1 41. Rd3 Rh5 42. Rd7 Rxg5+ 43. Kf1 Rh5 44. Kg1 Rc1 45. Rd5+ Kxe4 46. Rd7 Kf5 47. Rd5+ Ke6 48. Rd3 Ra1 49. Rd8 Rh3 50. Rd3 g5 51. Rd8 g4 52. Rd5 g3 53. fxg3 Ra2 54. Rh5 Rxh5 55. Bxf3 Rg5 56. g4 Rb2 57. Bxb7 Rxg4+ 58. Kf1 Rg3 59. Be4 Ke5 60. b4 Ra3 61. Bc2 Rxc2 62. bxc5 Ra1#

Checkmate 0-1

[*Event* "*ComputerChessGame*"]
[*Site* "*ALICE*"]
[*Date* "2013,11,07"]
[*Round* "−"]
[*White* "*Joli* − 1101"]
[*Black* "*Joli* − 1001"]
[*Result* "0 − 1"]
[*TimeControl* "300"]
[*Annotator* "2. + 0,022... − 0,26"]

1. Nf3 Nf6 2. d4 d5 3. Nc3 e6 4. e3 Bb4 5. Bd2 O-O 6. Bd3 Nc6 7. O-O Bd7 8. Ne5 Be8 9. Qf3 Qe7 10. Rad1 Bd6 11. Nxc6 Bxc6 12. Nb5 Bb4 13. Nxc7 Bxd2 14. Nxa8 Ba5 15. c3 Rxa8 16. b4 Bc7 17. b5 Bd7 18. c4 e5 19. dxe5 Qxe5 20. Qg3 Qxg3 21. hxg3 dxc4 22. Bxc4 Re8 23. Rd2 Kf8 24. Rfd1 Bg4 25. Be2 Ba5 26. Rc2 Bf5 27. Rc4 Be6 28. Ra4 Bb6 29. Bf3 Bd7 30. Rb4 Bc5 31. Rb2 a6 32. bxa6 bxa6 33. Be2 a5 34. Rb7 Ba4 35. Re1 Bc6 36. Rc7 Bb4 37. Rd1 Bd5 38. Bb5 Rd8 39. a4 g5 40. Ra7 Rd6 41. Be2 Kg7 42. Bf3 Kg6 43. Bxd5 Rxd5 44. Rxd5 Nxd5 45. Ra8 Nc3 46. f4 h6 47. Kf2 g4 48. Rg8+ Kh5 49. Rd8 f5 50. Re8 Ne4+ 51. Ke2 Nc5 52. Rf8 Kg6 53. Rg8+ Kf7 54. Rh8 Kg7 55. Rc8 Nxa4 56. Rc7+ Kg6 57. Rc6+ Kg7 58. Rc7+ Kf8 59. Rc6 Nc3+ 60. Kd3 Ne4 61. Rxh6 Ke7 62. Kc4 Bc5 63. Rh7+ Kf6 64. Kd3 Ke6 65. Rh6+ Kd5 66. Rh5 Ke6 67. Rh6+ Kd7 68. Ra6 Bb4 69. Kd4 Nxg3 70. Kd5 Ke7 71. Re6+ Kf7 72. Rc6 Bd2 73. Rc7+ Kf6 74. Rc6+ Ke7 75. Rb6 Kd8 76. Kd4 Kd7 77. Rh6 Ke7 78. Rh7+ Ke6 79. Rh6+ Kd7 80. Ra6 Ke7 81. Rh6 Kd7 82. Ra6 Ke7 83. Rh6 Kf7 84. Rc6 Ne4 85. Ke5 Ng3 86. Kd4 Bb4 87. Kd5 a4 88. Rc4 Bd2 89. Kd4 Nf1 90. Kd3 a3 91. Ke2 Nxe3 92. Rc7+ Ke6 93. Kxd2 Nxg2 94. Ra7 Nxf4 95. Rxa3 Nd5 96. Kd3 Kd6 97. Kc4 Ne7 98. Ra6+ Ke5 99. Ra1 f4 100. Re1+ Kf6 101. Kd4 Nf5+ 102. Kd5 f3 103. Re4 g3 104. Rf4 g2 105. Ke4 Ke6 106. Rxf5 f2 107. Rxf2 g1=Q 108. Rf3 Qg4+ 109. Ke3 Ke5 110. Kf2 Qd4+ 111. Ke2 Qc4+ 112. Kd2 Qb4+ 113. Ke2 Qb2+ 114. Kd1 Qd4+ 115. Ke2 Qb2+ 116. Kd1 Qb1+ 117. Kd2 Qb4+ 118. Ke2 Qg4 119. Kf2 Kd4 120. Rf7 Qg5 121. Rd7+ Ke4 122. Rf7 Qc5+ 123. Kg3 Qd6+ 124. Kf2 Qb6+ 125. Kf1 Qa6+ 126. Ke1 Qa1+ 127. Ke2 Qa2+ 128. Kd1 Qxf7 129. Kc2 Kd4 130. Kb2 Qc4 131. Ka3 Qb5 132. Ka2 Kc3 133. Ka3 Qb3#
Checkmate 0-1

Apéndice B

Joli - Manual de usuario

Este pequeño manual pretende ser una guía para la ejecución de Joli, tanto por consola como mediante la interfaz gráfica Winboard.

B.1. Archivo de configuración config.ini

El archivo de configuración config.ini contiene la inicialización de valores de Joli. A continuación se detallan las entradas que contiene.

- **time**: tiempo máximo de búsqueda (en segundos), por defecto está configurado en 60.
- **depth**: profundidad máxima de búsqueda (en plies), por defecto está configurado en 8.
- **STOP_FRAC**: Límite de tiempo para iniciar una nueva iteración; no iniciar una nueva iteración si ya hemos consumido este porcentaje de tiempo de búsqueda, configurado en 70 %.
- **DEPTH_MATE**: Profundidad de búsqueda para la función mateIn-N. el valor por defecto es 2, para deshabilitar la funcionalidad del MateIn-N ingresar el valor 0.
- **TIMER_MATE**: fracción del tiempo de búsqueda de la posición asignado a la búsqueda del mateIn-N, el valor por defecto es 50 %, si se ingresa 0 entonces se deshabilita la ejecución de esta función

- **EVAL_FUNC**: Parámetro para seleccionar la función de evaluación, si se elije 1, se utilizará la nueva versión implementada para "Joli"; si se ingresa 0 se utilizará la del motor original "Winglet".
- **PARAM_EVAL_MATERIAL**: coeficiente multiplicador para la evaluación material del tablero; si se carga en 0 se deshabilita
- **PARAM_EVAL_ESPACIAL**: coeficiente multiplicador para la evaluación espacial del tablero; al igual que el anterior, su valor en 0 hace que se deshabilite
- **PARAM_EVAL_DINAMICA**: coeficiente multiplicador para la evaluación dinámieca del tablero, si se pone en 0 se deshabilita
- **PARAM_EVAL_POS_TABLERO**: coeficiente multiplicador para la evaluación posicional del tablero, su valor en 0 indica que está deshabilitada.

Es importante destacar que estos últimos cuatro coeficientes deben tener valores enteros.

B.2. Iniciando la aplicación en modo consola

Para iniciar la aplicación por consola, solamente es necesario hacer "doble click"sobre el archivo "Joli.exe". Es importante tener en cuenta que se debe contar con las dlls msvcr110.dll y msvcp110.dll en la misma carpeta de la aplicación; esto para prevenir posibles errores con las versiones del redistribuíble MS Visual C++.

Lista de comandos por consola:

- **help** ó **h** ó **?**: muestra el listado de comandos aceptados, junto con una breve descripción de los mismos
- **black**: se pasa el turno al jugador negro
- **cc**: hace que juegue Joli con ambos colores
- **d**: despliega el tablero actual

- **eval**: muestra una evaluación estática de la posición actual (evalúa el tablero)
- **exit** ó **quit**: finaliza la ejecución de la aplicación
- **game**: muestra todos los movimientos realizados hasta la posición actual
- **go**: le cede el control a Joli para que realice el próximo movimiento
- **info**: despliega variables del tablero (para propósitos de testeo).
- **ini**: reinicializa las variables internas con la información dispuesta en el archivo "config.ini"
- **memory n**: máxima memoria que se le permitirá usar, en MB; ejemplo: memory 15
- **move orig/dest**: mover la pieza en el casillero "orig" al casillero "dest"; ejemplo move e2e4 o move h7h8q. No es necesario poner la pieza que estamos moviendo, solamente el casillero. Para el caso de una coronación de un peón, necesita indicarse también la pieza en la que se transformará con la letra correspondiente; para reina poner "q", para caballo "n", para torre "t" y para alfil "b" para finalizar el comando de movimiento
- **moves**: despliega por pantalla todos los movimientos legales posibles
- **new**: inicia una nueva partida
- **perft n**: calcula el número total de nodos a partir de la posición actual, hasta una profundidad n que se le indique; ejemplo: preft 3
- **r**: rota el tablero
- **readfen filename n**: carga del archivo "filename" la posición número n. Es importante destacar que la ruta del archivo debe ser completa (por ejemplo "C:\repositorioPartidasFen\partidas.fen")
- **sd n**: actualiza con el valor n la profundidad máxima de búsqueda
- **setup**: configura el tablero

- **test filename**: inicia una búsqueda en cada una de las posiciones FEN descritas en filename usando los parámetros de tiempo y profundidad que tenga actualmente definidos. El resultado se escribe en el archivo test.log
- **time n**: actualiza el tiempo máximo, en segundos, a utilizar en la búsqueda con el valor n
- **undo**: deshace el último movimiento
- **white**: se pasa el turno al jugador blanco
- **isCheck**: retorna un mensaje indicando si hay o no jaque
- **isMate**: retorna un mensaje indicando si hay o no jaque mate
- **#moves**: despliega el número total de movimientos disponibles
- **castles**: indica qué enroques están disponibles para el jugador de turno
- **material**: despliega el valor total de las piezas para cada jugador
- **dup**: muestra las jugadas duplicadas para el jugador de turno
- **movSPawn**: muestra el número de jugadas sin mover o tomar un peón
- **reachables**: muestra el número de casilleros a los que puede acceder el jugador de turno
- **enpassantMoves**: muestra las capturas al paso posibles para el jugador de turno
- **readpgn filename n**: carga el tablero a partir de un archivo PGN con la lista de movimientos de la partida. El valor n es opcional e indica hasta que jugada cargar
- **mateIn filename numeroFen profundidad**: chequea si para el tablero en la posición numeroFen en el archivo filename, el jugador de turno puede forzar una secuencia de mate en la profundidad indicada de movimientos.
- **think**: invocación a la función de evaluación para que nos indique su mejor movimiento.

B.3. Iniciando la aplicación desde Winboard

Para poder utilizar Joli con la interfaz gráfica Winboard, primero debemos cargar el motor y luego seleccionarlo de la lista de motores activos. A continuación se muestran los pasos que debemos realizar.

1. Abrir la interfaz Winboard y seleccionar del menú la opción "Engine" y luego "Load First Engine"

Figura B.1: Opción de menú

2. Se nos desplegará la siguiente ventana, en la que deberemos adjuntar el ejecutable de nuestro motor (Joli.exe) en el campo “Engine (*.exe)”. Los demás campos pueden quedar con los datos que trae por defecto. En caso que ya hayamos cargado a Joli previamente, solamente debemos seleccionarlo de la lista de motores que se encuentra a la izquierda.

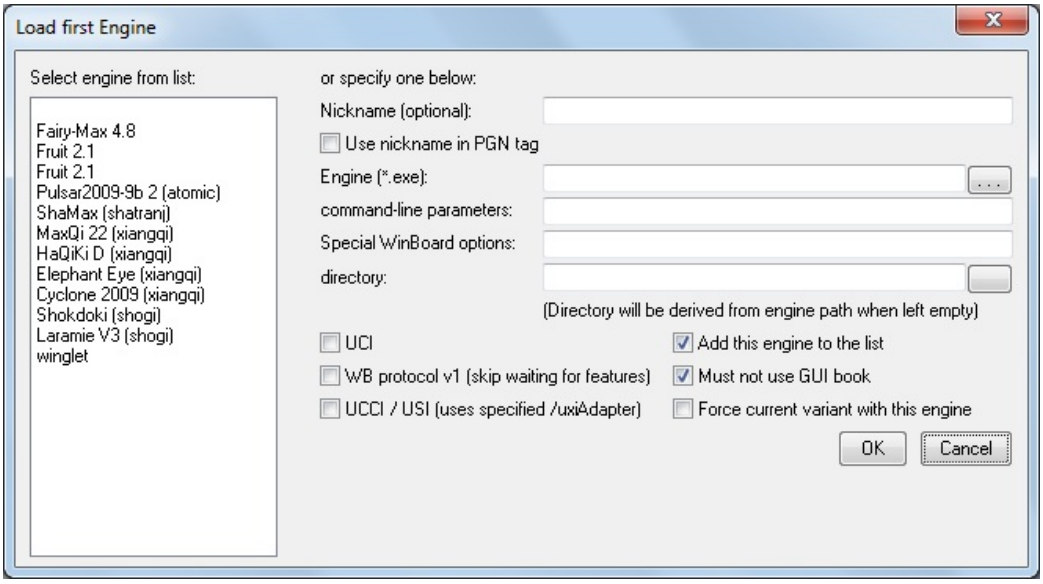

Figura B.2: Seleccionar motor

3. Damos click en “OK”

4. Luego, seleccionamos la opción de menú “Mode” e indicaremos con las opciones “Machine White” o “Machine Black”si queremos que el motor juegue de Blancas o Negras y se iniciará automáticamente la partida.

Figura B.3: Selección del bando