UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

INTELIGENCIA ARTIFICIAL APLICADA AL ESTUDIO DEL AJEDREZ HEXAGONAL

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A:

CARLOS BADILLO LORA

DIRECTOR DE TESIS:
DRA. KARLA RAMÍREZ PULIDO

Ciudad Universitaria, CD. MX. 2024

ÍNDICE GENERAL

CAPÍTULO 1

INTRODUCCIÓN

En la actualidad el hacer uso de juegos a través de diferentes medios tecnológicos es una actividad que muchas personas realizan en algún momento de su vida. ¿Cómo funcionan estos juegos? Algunos de ellos simplemente usan una rutina pre programada o un árbol de decisiones para reaccionar al usuario. Otros juegos usan una variedad de técnicas de IA como los árboles de juego, computación evolutiva o redes neuronales. Estás técnicas permiten al jugador disfrutar de una experiencia más interesante y personalizada.

A lo largo de la historia de la Inteligencia Artificial los juegos se han beneficiado de los avances en esta rama, y a su vez han servido de catalizadores para realizar nuevos descubrimientos.

Entre las razones por las que los juegos son objetos de estudio interesantes para la Inteligencia Artificial podemos mencionar en primer lugar el hecho de que un juego es fácil de modelar. En segundo lugar podemos tener el control de todas las variables y se rigen por reglas simples. En tercer lugar porque son difíciles de jugar bien. Crear un programa que juegue puede ser una tarea complicada debido a la gran cantidad de estrategias que se pueden usar en ellos.

Aunque los juegos en general son interesantes para el estudio en inteligencia artificial; pues nos presentan problemas complejos e interesantes [YT18, pág. 4]. El ajedrez nos ofrece una ventaja adicional: podemos medir fácilmente el desempeño de un programa enfrentándolo con otros programas o incluso con un humano.[New75, pág. 2]

Algunos de los hitos más conocidos en la historia de la inteligencia artificial están relacionados con los juegos. Como la famosa victoria de *Deep Blue* sobre el que era considerado el mejor ajedrecista, Gary Kasparov [YT18, pág. 8], o la derrota del jugador número 1 de *Go*, Ke Jie, frente a *AlphaGo* [YT18, pág. 9].

Los fundamentos del ajedrez hexagonal son iguales a los del ajedrez tradicional:

- Es un juego por turnos entre dos jugadores.

- Es de "*suma cero*". Esto significa que no hay movimientos cooperativos, cada movimiento daña a un jugador tanto como beneficia a su rival.

Es un juego de "*información perfecta*" lo que significa que los estados de todas las situaciones del juego son completamente visibles, a diferencia de un juego como el póquer.

Sin embargo su complejidad es más alta que la del ajedrez convencional debido a que:

- El movimiento de las piezas a lo largo de casillas con seis vecinos aumenta el número de movimientos.
- Otro factor que aumenta la complejidad del juego es el hecho de que tiene un tablero más grande y de tres colores.
- El tener tres colores nos permite tener tres *alfiles* por jugador, lo que aumenta las estrategias posibles.

El objetivo principal de este trabajo de titulación es crear un programa de ajedrez hexagonal eficiente y retador para un jugador humano de nivel principiante usando la heurística *Minimax*.

Los objetivos secundarios son:

- Estudiar el uso de la inteligencia artificial aplicada a una variante del ajedrez.
- Analizar y comparar distintas técnicas y heurísticas.
- Crear una interfaz de usuario para el juego.

Este trabajo de titulación representa un caso interesante en el área de la Inteligencia Artificial, pues se ponen a prueba los métodos que se usan en el ajedrez clásico aplicándolas en una variante más compleja con el propósito de estudiar su rendimiento. Además este trabajo de titulación también beneficia a los jugadores de ajedrez pues hasta el momento existen pocos programas de *software* libre dedicados a este juego. Se desarrollaron dos versiones del juego, una en *Python* y otra en C++. Al final se ofrece una comparativa entre ambas, con un algoritmo de poda y con *Minimax* simple.

En resumen en este proyecto se desarrolla un programa que usa la heurística *Minimax* para jugar el juego de Ajedrez Hexagonal además de una interfaz de usuario que se desarrolló usando las bibliotecas *Pygame* para *Python* y *SDL* de C++.

El primer capítulo consiste de la introducción de este trabajo, para exponer el contexto del mismo. El segundo capítulo está dedicado a la historia del ajedrez como juego y también como área de investigación en Inteligencia Artificial. También se describe la variante del ajedrez hexagonal y se da una breve historia. Después se pasa a describir otros proyectos de *software* similares al propuesto en esta tesis. El tercer capítulo está dedicado a las heurísticas y técnicas usadas para el desarrollo de programas capaces de jugar juegos de suma cero e información perfecta. Aquí se abordan las distintas técnicas y tecnologías que se han utilizado para este tipo de programas. En el capítulo cuatro se describe el funcionamiento del motor de ajedrez hexagonal y por último se describe cómo funciona la interfaz de usuario. En el quinto capítulo se exponen las conclusiones de este trabajo y el trabajo a futuro.

CAPÍTULO 2

HISTORIA

2.1. Historia del Ajedrez

En esta sección la información es tomada principalmente del libro *Chess. The History of a Game* de Richard Eales.

La teoría más aceptada sobre los inicios del ajedrez nos dice que este surgió en el norte de la India. Una teoría menos probable sugiere que se originó en China [Eal85, págs. 33-35]. Es difícil determinar cuándo se inventó el ajedrez. La evidencia más temprana del juego data de después del año 600, por lo tanto podemos especular que el ajedrez apareció en el siglo VI, aunque bien pudo haber aparecido antes [Eal85, pág. 36].

Desde antes de la era cristiana ya existían varios juegos de mesa en el norte de la India [Mur12, pág. 47]. Ahora bien, existe un texto escrito entre el II a.C. y el inicio de la era cristiana: El *Mahabhashya* de Patañjali. Este texto es un comentario sobre una gramática más antigua, que contiene una descripción detallada del "*Ashtapada*", un tablero de juego cuadrado dividido en 64 casillas [Mur12, pág. 48]. ¿Qué clase de juego se jugaba en el "*Ashtapada*", la evidencia textual nos indica que se jugaba un juego de dados, donde se movían piezas rojas y negras [Mur12, pág. 52]. Es razonable suponer que este era un juego de carreras, parecido al *Backgammon* moderno.

La teoría desarrollada por el historiador H.J.R. Murray es que el Ajedrez surgió cuando alguien usó el tablero "*Ashtapada*" para hacer un juego de guerra. Aunque esta teoría es muy convincente, no ha sido probada de forma definitiva [Eal85, pág. 31]. Una teoría alterna es que surgió como una práctica adivinatoria. Se usaría un tablero, dados y piezas representando soldados para poder profetizar el curso de una batalla. En algún momento alguien decidió retirar el dado y convertir esta práctica religiosa en un juego [HW92, pág. 173].

El nombre en sánscrito del ajedrez es *chaturanga*. Esta palabra está formada por las palabras *chatu* (cuatro) y *anga* (miembro). Entre otros usos, esta palabra se usaba para designar a un ejército y es que los ejércitos en la India antigua estaban formados por cuatro partes. Infantería, Caballería, Carros de Guerra y Elefantes [Mur12, págs. 62-63]. Todas estas fuerzas están representadas en el ajedrez primitivo [Mur12, pág. 64].

La primera referencia al ajedrez en la literatura ocurre en el *Harschacharita*. Esta es una biografía del emperador indio Harshah y fue escrita por el poeta Bana, aproximadamente en el año 640 [CK17, pág. 50].

Desde la India este juego se diseminó hacia China; donde se adaptó al carácter nacional y de aquí se extendió hacia Corea y Japón. En Japón se modificó aún más el juego y se convirtió en el *Shogi* Japonés.

Del lado opuesto del subcontinente; el juego se difundió desde la India hacia Persia alrededor del año 625 [HW92, pág. 173]. Ahí se le dio el nombre de *Chatrang* [HW92, pág. 173]. Es del lenguaje persa de dónde se derivan los nombres del juego en las lenguas europeas (*Chess, Schach, Échecs*, Ajedrez, etc) [Mur12, pág. 40], así como los nombres de las piezas y la expresión "*Jaque Mate*". El juego de *Chatrang* tenía 64 casillas y 32 piezas. 16 piezas color esmeralda y 16 color rubí. Las piezas eran 1 *rey*, 1 ministro, 2 elefantes, 2 *caballo*s, 2 carros de guerra y 8 soldados de infantería [She11, pág. 22].

En el año 638 el Califa Umar ibn al-Jhattab comenzó la conquista de Persia, que se terminaría de consumar en 651. Uno de los resultados de esta conquista fue la introducción del ajedrez al mundo árabe [Mur12, pág. 278]. Debido a las estrictas reglas contra las imágenes, los musulmanes sunitas abandonan las figurillas y las reemplazan por piezas abstractas. Los musulmanes chiítas continuaron usando las piezas representativas [Mur12, págs. 281-282]. En el mundo musulmán el ajedrez llegó a un alto grado de sofisticación y es donde encontramos los primeros textos dedicados exclusivamente al ajedrez, en los que podemos encontrar registros de partidas, análisis de aperturas y problemas [Eal85, pág. 20]. Un ejemplo de esto es el primer libro dedicado al análisis del ajedrez, *Kitab ash-shatranj* (el libro del ajedrez) escrito por al-Adli, el mejor jugador de su tiempo [She11, pág. 33].

Irónicamente, el ajedrez no llegó a Europa occidental a través de los cristianos bizantinos [Eal85, pág. 41]. La evidencia etimológica nos dice que fueron los árabes musulmanes los que llevaron el ajedrez a Europa. El juego se introdujo a través de las penínsulas Ibérica e Itálica, lugares donde había frecuentes intercambios culturales entre musulmanes y cristianos [Eal85, pág. 48]. En Europa se reemplazaron las casillas monocromáticas usadas en Asia por el distintivo patrón de blanco y negro. Además los europeos dejaron las piezas abstractas y retomaron las figuras representativas [She11, pág. 44].

El juego llegó a Rusia a través de tres rutas: la primera fue a través de los comerciantes del Volga, la segunda a través de bizantinos que lo llevaron a los Balcanes y la tercera fueron los vikingos en el báltico [HW92, pág. 173].

Para el final del primer milenio ya se jugaba ajedrez en toda Europa. Al inicio hubo algo de oposición al juego por parte de las autoridades eclesiásticas. Sin embargo, hubo dos razones por las que el juego no fue censurado. La primera era su gran popularidad entre la nobleza e incluso entre el clérigo. La segunda fue el argumento de que el ajedrez, al ser un juego de habilidad y no de azar, no constituía un vicio [Eal85, pág. 60].

Durante la edad media el ajedrez se convirtió en parte de la educación de la nobleza [Eal85, pág. 53]. También era popular en las universidades y conventos. No tuvo tanta aceptación entre las clases bajas, que preferían los juegos de mesa autóctonos. El nivel de juego era por lo general bajo [Eal85, pág. 69] y normalmente se jugaba por una apuesta [Eal85, pág. 55].

En el medioevo hubo un gran número de trabajos dedicados a explicar el ajedrez de forma alegórica. Estos trabajos reciben el nombre de "Moralidades" [Mur12, págs. 807-808]; muchas veces los autores no estaban realmente preocupados por el juego en sí, sino que lo usaban como herramienta para hablar sobre la sociedad [Eal85, pág. 64]. El más notable de estos libros fue *Liber de moribus hominum et officis nobilium* (El libro de las costumbres de los hombres y obligaciones de los nobles) escrito por fray Jacobus de Cessolis [Eal85, pág. 66]. En ésta época el prestigio del juego era social más que intelectual [Eal85, pág. 69].

En el último cuarto del siglo XV surge una nueva variante del ajedrez conocida como "*alla rabiosa*", "*sacchi de la donna*", "*axedrez de la dama*" o "*eschés de la dame*" [Mur12, pág. 1265]. Este juego difería del "*axedrez del viejo*" en los movimientos de la *dama* y del *alfil*. Estas piezas perdieron la habilidad de saltar sobre una casilla ocupada pero en cambio el *alfil* ganó la habilidad de moverse a cualquier casilla en diagonal, siempre y cuando el camino esté libre. La dama ganó los movimientos del nuevo *alfil* y de la *torre*. El resultado fue que estas piezas se fortalecieron considerablemente. Antes la dama era más débil que la *torre* y el *caballo*, era tan solo un poco más poderosa que el *alfil* [Mur12, pág. 1265]. Pero ahora se volvió la pieza más poderosa del tablero. El fortalecimiento de la *dama* también reforzó al *peón*, pues ahora se podía convertir en una pieza más valiosa al coronarse.

No se sabe a ciencia cierta si estas nuevas reglas surgieron en Francia, España o Italia. Pero la evidencia que apoya el caso Italiano es la más fuerte [Mur12, pág. 1266]. Con el tiempo estas nuevas reglas reemplazaron al "*axedrez del viejo*" y se convirtieron en estándar.

Estos cambios hicieron que las partidas fueran más rápidas y también hizo al juego más difícil, cualquier error podía costar la partida [Eal85, pág. 77]. Esto hace necesario analizar las distintas formas de empezar una partida [Mur12, pág. 1265], es el nacimiento de la teoría de aperturas.

Los primeros escritores y teóricos de este nuevo ajedrez provinieron de la península ibérica, por ejemplo: Vicent, Lucena, Damiano y Ruy López [HW92, pág. 174] quien le dio nombre a la apertura más famosa del juego.

El periodo comprendido entre la publicación del libro de Ruy López y la mitad del siglo XVII es conocido como la "era heróica". En la que los jugadores italianos dominaron el juego, el más famoso de ellos fue Gioachino Greco "*il calabrese*" [Eal85, págs. 83, 86] sus manuscritos fueron muy notables y sirvieron de referencia durante más de un siglo. En Francia su "Juego del Ajedrez" sirvió hasta mediados del siglo XVIII [Eal85, pág. 98].

El siglo XVII fue uno de transición para el ajedrez. El aumento en la dificultad del juego y la erudición que requería dominarlo redujeron su atractivo tradicional [Eal85, pág. 94]. Dejó de ser parte indispensable de la educación del caballero; que ahora tenía un abanico mucho más amplio de actividades esparcimiento [Eal85, pág. 106]. Pero en cambio surgió un nuevo tipo de jugador que se sentía atraído por sus cualidades intelectuales.

Este nuevo jugador no jugaba meramente para cumplir una expectativa social, sino por interés genuino [Eal85, pág. 106].

A mediados del siglo XVIII Italia perdió su prominencia y el centro del juego se movió a Francia e Inglaterra [Mur12, pág. 1383]. Tanto en París como en Londres los jugadores se reunían regularmente en Cafés para jugar y discutir. Los más importantes eran el *Café de la Régence* y el *Slaughter's Coffee House* [Eal85, pág. 109].

El más prominente jugador del siglo XVIII fue el compositor de música André Philidor (n. 1726 , m. 1795), quien dominó tanto París como Londres durante 40 años [Mur12, pág. 1370] cautivó a la sociedad de su tiempo al jugar 3 partidos simultáneos con los ojos vendados, algo inaudito para la época [She11, pág. 71]. Le daba una gran importancia a la estructura de los *peones*, llegando a declarar que los *peones* son "el alma del ajedrez" [She11, pág. 71]. Su libro *L'analyze des échecs* no sólo fue un éxito instantáneo sino que además tuvo influencia duradera [Mur12, pág. 1374].

A inicios del siglo XIX el liderazgo ajedrecista se lo disputaban Londres y París. París tuvo la ventaja primero cuando La Bourdonnais venció a McDonell en 1834 [Eal85, pág. 133]. Este duelo causó gran furor e incluso llegó a inspirar poemas como "*Une revanche de Waterloo*" de Joseph Méry [Mur12, pág. 1394]. Marcó la cúspide de la llamada "era romántica".

París siguió siendo la capital del ajedrez hasta 1843; año en el que Howard Staunton venció a Pierre C. F. de Saint-Amant. A partir de este momento Inglaterra le arrebató a Francia la preponderancia en el ajedrez que ésta había ostentado desde la época de Philidor [Mur12, pág. 1399].

Staunton contribuyó mucho al progreso del ajedrez: fundó clubes, intentó estandarizar las reglas internacionales y promovió un diseño de piezas estándar [HW92, pág. 174]. Fue él quien organizó el primer torneo internacional en Londres en 1851, que coincidió con la Gran Exhibición ese mismo año [She11, pág. 82].

Durante la segunda mitad del siglo XIX proliferaron los clubes de ajedrez. Este fenómeno empezó en Inglaterra, pero luego se extendió a Holanda, Estados Unidos y especialmente a Alemania [Eal85, pág. 142], de donde vinieron varios de los mejores jugadores de la época.

También en la segunda mitad del siglo XIX se popularizaron los torneos con recompensas que le dieron a los jugadores una fuente de ingresos además de dar lecciones. Esto contribuyó a la profesionalización del juego. Para estos tiempos el ajedrez ya se había consolidado como un pasatiempo de las clases medias [Eal85, pág. 139].

Las últimas décadas del siglo XIX fueron dominadas por Wilhelm Steinitz. Stenitz era judío bohemio, pero su carrera se desarrolló en Inglaterra y más tarde en Estados Unidos [Mur12, pág. 1403]. Fue campeón del mundo entre los años 1866 a 1894. Es considerado el fundador de la escuela científica o moderna. Este estilo de juego es cauteloso y defensivo y contrasta con la escuela romántica [She11, pág. 117]. La escuela científica extendió su dominio hasta las primeras décadas del siglo XX con jugadores como Lasker y Tarrasch.

La Primera Guerra Mundial afectó de forma negativa al juego, especialmente en países de Europa central como Alemania, debido a la carencia de torneos y oportunidades para los jugadores profesionales [Eal85, pág. 158].

En los años 20 surgió la escuela hipermoderna [Eal85, pág. 164]. Esta escuela enfatiza el dinamismo, prefiere ocupar los flancos con *caballo*s y *alfiles* al inicio del juego, en vez de intentar ocupar el centro con *peones* [She11, pág. 129]. Sus principales proponentes fueron Aron Nimzowitsch, Richard Réti y Gyula Breyer [HW92, pág. 361].

Fue también en esta década cuando el gobierno de la Unión Soviética comenzó a invertir fuertemente en el juego [Eal85, pág. 169]; aunque no sería sino hasta después de la guerra cuando empezaría a cosechar victorias. Los años 20 también fueron los del prodigio Raúl Capablanca. En 1921 el cubano se convirtió en campeón del mundo al vencer a Lasker en en la Habana [HW92, pág. 68].

En 1927 fue creada la *Fédération Internationale des Échecs*, *FIDE*, una organización que marcó la pauta del juego durante el siglo XX. En 1927 organizó la primera olimpiada y el primer campeonato mundial femenil [HW92, pág. 174]. Y a partir de 1948 comenzó a conferir el título de campeón mundial

En la segunda mitad del siglo XX el dominio lo adquirió la Unión Soviética, esto se debió en gran medida al gran impulso que tuvo el ajedrez por parte del gobierno. El dominio soviético empezó con el campeonato de 1948 en Moscú y La Haya en el que Mikhail Botvinnik se estableció como campeón mundial [TG07, pág. 8]. Botvinnik fue campeón hasta 1963, con dos breves interrupciones donde Smyslov y Tal fueron campeones. Después fue sucedido por Tigran Petrosian, y este fue sucedido por Boris Spassky. Todos ellos ciudadanos soviéticos.

El reinado soviético no sólo fue mantenido por jugadores brillantes y amplio apoyo del estado; sino también por tácticas deshonestas. Por ejemplo, en el campeonato mundial de 1963 los jugadores soviéticos habían acordado empatar partidas entre ellos.

En 1971 el americano Bobby Fisher desafió la preponderancia soviética y calificó para disputarle el título a Spassky [She11, pág. 119]. El año siguiente se jugó la llamada "partida del siglo" en la que Fisher derrotó a Spassky y rompió el dominio soviético.

Poco tiempo después de su victoria Fisher se retiró del ajedrez profesional y la Unión Soviética retomó su principalidad hasta la disolución de la propia Unión Soviética.

Las últimas décadas del siglo XX fueron dominadas por dos grandes maestros de la escuela soviética: el ruso Anatoly Karpov y el azerbaiyaní Gary Kasparov. Karpov fue campeón de la *FIDE* desde 1975 hasta 1985 cuando fue derrotado por Kasparov quién fue campeón hasta 1993.

En 1993 hubo un cisma en el mundo del ajedrez cuando Kasparov y su contrincante Nigel Short decidieron disputar el título de campeón mundial fuera de la *FIDE*. Esta respondió quitándole a Kasparov su título, y proclamando a Karpov como campeón mundial después de que derrotara al neerlandés Jan Timman. Por su parte Kasparov y Short formaron la Asociación Profesional de Ajedrez. donde Kasparov se convirtió en campeón después de derrotar a Short [Den08, pág. 9].

A pesar de la disolución de la Asociación Profesional de Ajedrez en 1995 y el retiro de Kasparov en 2005 [TG07, pág. 17] el cisma no se resolvió sino hasta el año 2006 cuando Kramnik y Topalov jugaron por el título unificado. Kramnik fue sucedido por el indio Viswanathan Anand. La década de los 2010s fue dominada por el prodigio danés Magnus Carlsen.

2.2. Historia del Ajedrez de Computadora

El interés por una computadora que pudiera jugar ajedrez estuvo presente desde los inicios de la computación. En 1864 Charles Babbage especuló sobre la posibilidad de que su motor analítico pudiera jugar ajedrez. Para él, el mayor problema sería poder representar los innumerables estados del juego [New75, pág. 6]. Aunque no fue más allá con el ajedrez, sí describió un autómata que podría jugar *tic tac toe* [Mon13, pág. 2].

La primera máquina electromagnética capaz de jugar ajedrez de forma limitada fue "El Ajedrecista", creada por Leonardo Torres y Quevedo alrededor de 1890. Esta máquina era capaz de jugar un final de partida contra un humano, Las piezas con las que se jugaba eran *rey* y *torre* contra *rey* [RN95, pág. 142].

En 1928 John von Neuman publicó *Zur Theorie der Gesellschaftspiele* donde investiga lo que denomina "juegos sociales" y qué estrategias deben usar los jugadores para obtener el mejor rendimiento posible [Neu28, pág. 95]. En este artículo establece las bases del método *Minimax*, [Her18, pág. 162]. En los 40 von Neuman retomó el estudio de los juegos y, en colaboración con el economista Oskar Morgenstern, escribió el libro *Theory of Games and Economic Behaviour*, publicado en 1944. En este libro explican cómo se puede usar el algoritmo *Minimax* para jugar ajedrez.

El siguiente paso lo dio Claude Shannon en 1950 cuando publicó el artículo fundamental del ajedrez de computadora: *Programming a Computer for Playing Chess* Aquí detalla más como se podría usar un árbol de búsqueda para jugar ajedrez. Shannon propone darle a cada posición del tablero una evaluación basada en el material, en la estructura de los *peones* y en la movilidad [New75, pág. 9]. Adicionalmente describe dos estrategias distintas para limitar el árbol de búsqueda. La "estrategia tipo A" es simplemente limitarlo sin ninguna consideración a una profundidad uniforme. La "estrategia tipo B" [1] consiste en usar heurísticas para descartar las ramas poco interesantes [Ens11, pág. 11].

Mientras tanto Alan Turing también se interesó por el ajedrez de computadora. Al igual que Shannon propuso el uso del algoritmo *Minimax* y una función de puntaje, aunque su función es más simple y está basada sólo en el material. En 1951 se jugó el primer juego entre un humano y un algoritmo. El algoritmo *Minimax* fue ejecutado por el propio Turing [New75, págs. 15-16].

Las ideas de Shannon y Turing se pusieron en práctica en 1957 cuando se hizo el primer programa capaz de jugar una versión del ajedrez en Los Álamos por un equipo que incluía a James Kister, Paul Stein, Stanislaw Ulam, William Walden y Mark Wells. El programa estaba escrito para la computadora *MANIAC I* y no jugaba ajedrez clásico sino una variante reducida llamada "ajedrez miniatura". El ajedrez miniatura se juega en un tablero de 6 por 6 casillas. No tiene *alfiles* y cada jugador solo tiene 6 *peones* [New75, pág. 19].

[1] Las estrategias basadas en *Minimax* se pueden clasificar en estrategias de Tipo A y Tipo B. La diferencia es sobre si todos los caminos son buscados a la misma profundidad o por el contrario se van a elegir algunos caminos para buscarse de forma más exhaustiva. Esta distinción fue introducida por Shannon en 1949 [Her18, pág. 163]

Las estrategias tipo A están basadas en la búsqueda *Minimax* de profundidad uniforme y anchura completa [Abr89, pág. 143]. Esto es comparativamente fácil de implementar y conceptualmente simple de justificar, conforme los estimados se acercan a un valor exacto el procedimiento se acerca al desempeño óptimo. Sólo hay errores en la evaluación estática. La principal desventaja de estas estrategias es que necesitan de una elevada cantidad de computo [Abr89, pág. 143].

Las estrategias de tipo-B tienen como única característica común que buscan expandir solamente las líneas de juego más prometedoras [Abr89, pág. 7].

En 1955 se escribe el primer programa jugador de ajedrez en un lenguaje de alto nivel. El programa es obra de Alan Newell, John Shaw y Herbert Simon y se escribió en el lenguaje IPL-IV, creado por los mismos autores del programa [New75, pág. 25].

Mientras tanto, en la Unión Soviética el estudio del ajedrez de computadora se centraba en Moscú, en el Instituto para la Física Experimental, *ITEP* por sus siglas en inglés. Aquí era donde George M. Adelson Velsky dirigía el desarrollo de un programa de Ajedrez para la computadora M-20. Este programa seguía la estrategia tipo A de Shannon [Her18, pág. 165], la cual se describe en la nota 1(Página 8).

En 1966 se llevó a cabo una partida de ajedrez de computadora entre la universidad de Stanford en Estados Unidos y el *ITEP*. Stanford fue representada por John McCarthy (creador de *LISP*) y Alan Kotok quienes enfrentaron su motor de ajedrez contra el del *ITEP* [New75, pág. 3]. La comunicación era por telégrafo, se jugaron 4 juegos y el ganador fue el *ITEP* [Her18, pág. 165].

Ese mismo año Richard Greenblatt del *Massachusetts Institute of Technology* introdujo las tablas de transposición que permite guardar posiciones evaluadas anteriormente en caché para no tener que evaluarlas otra vez [She11, pág. 148].

En la década de los 70 se implementó el método *Bitboard* para representar una partida de ajedrez. Esto se logró gracias a varias personas trabajando de forma independiente, entre ellos Adelson Velsky, Hans Berliner, Slate y Atkin [Her18, pág. 167]. También fue en esta década cuando surgieron los torneos de ajedrez de computadora como el *NACCC (North American Computer Chess Championship)* organizado por la *ACM* y el *WCCC* (*World Computer Chess Championship*) [Her18, pág. 167].

En el año de 1983 se logró otro hito en la historia del ajedrez de computadora: el primer programa oficialmente reconocido Maestro por la Federación de Ajedrez de Estados Unidos. El nombre del programa era *Belle* y fue creado Joe Condon y Kenneth Thompson [Her18, pág. 168]

En 1989 *IBM* contrató a tres investigadores de Carnegie Mellon con el propósito de construir una computadora que pudiera vencer al mejor jugador del mundo. Ellos eran Feng-Hsiung Hsu, Murray Campbell y Thomas Anantharaman [New03, pág. IX], aunque este último abandonaría el proyecto más tarde. El primer programa creado por este equipo fue *Deep Thought*.

Deep Thought compitió y ganó el sexto *WCCC, World Computer Chess Championship* (Campeonato mundial de Ajedrez de Computadora). Este se llevó a cabo en el año de 1989 en Hong Kong [New11, pág. 27]. Pero ese mismo año perdió contra Kasparov, campeón de la *FIDE*. En 1995 un prototipo llamado *Deep Blue Prototype* compitió en el octavo *WCCC* donde quedó en tercer lugar con Fritz ganando el primer lugar [New11, pág. 27]. El siguiente enfrentamiento entre *Deep Blue* y Kasparov fue en 1996 cuando jugaron un mini-torneo de seis juegos. Este torneo lo ganó Kasparov 4 a 2 [Woo21, pág. 75]. La revancha fue un año más tarde y esta vez *Deep Blue* resultó vencedor. Este último duelo estuvo sumido en controversia pues Kasparov sospechaba que *IBM* había hecho trampa [Woo21, pág. 75].

Kasparov tenía interés en jugar contra *Deep Blue* otra vez. Pero *IBM* decidió retirar el programa, *Deep Blue* fue desarmada y entregada a dos museos: El museo *Smithsonian* en Washington, D.C. y el C*Computer History Museum en Mountain View*, California [New11, pág. 4]. Con el retiro de *Deep Blue* se volvió a abrir

la pregunta sobre cuál era el mejor motor de ajedrez en el mundo. Esto se resolvería en el noveno *WCCC* donde el motor *Shredder*, desarrollado por Stefan Meyer-Kahlen resultó victorioso [New11, pág. 29].

En el 2010 surgió un nuevo torneo que opacó al *WCCC*: el *Top Chess Engine Championship* , *TCEC* (Campeonato del Mejor Motor de Ajedrez). Este torneo tiene varias ventajas sobre *WCCC*, la primera es que los motores compiten usando hardware muy similar. Esto garantiza que gane el motor mejor programado y no solamente la computadora más potente. Además usan los mismos libros de apertura. En el ajedrez de computadora un libro de aperturas es un conjunto de posiciones de tablero cuyos valores ya están calculados. Estas posiciones corresponden a las aperturas más comunes del ajedrez. Finalmente, en el *TCEC* los motores juegan cientos de juegos en vez del número limitado del *WCCC*.

En los últimos años el dominio de este torneo lo han tenido dos motores de código abierto: *Stockfish* y *Leela Chess Zero*. En el mundo del ajedrez de computadora, estos dos motores son el estado del arte[HH21, pág. 8].

2.3. Motores dominantes en la actualidad

A continuación se da una breve descripción de los motores de ajedrez dominantes en el momento. Existen dos familias notables de motores. Los motores tradicionales usan *Minimax* y el algoritmo de poda *Alfa-beta* . El motor más notable en esta categoría es *Stockfish*.

La otra familia de motores usa redes neuronales y el algoritmo Árbol de Búsqueda *Monte Carlo*. Esta familia se popularizó debido al éxito de *Alpha Go* y *Alpha Zero*. El más notable miembro de esta categoría es *Leela Chess zero* [Dog20].

El Maestro *FIDE* Bill Jordan dijo de *Stockfish* que "representa el cálculo" y de *Leela Chess Zero* que "representa la intuición" [MPT22, pág. 10].

2.3.1. Stockfish

Stockfish fue creado por Marco Costalba, Joona Kiiski, Gary Linscott y Tord Tomstad [Sre18, pág. 1]. Este proyecto comenzó en 2008 usando como base un programa anterior llamado *Glaurung*. Hoy en día tiene muchos contribuyentes. Usa un *bitboard* para representar el estado de la partida [Qui10]. Está escrito en C++ y es de código libre.

Stockfish usa un árbol de juego para representar las posiciones posibles del tablero. A cada posición se le asigna un vector de características que contiene cosas como el material, la posición, estructura de *peones*, etcétera. Estas características se agregan en una combinación lineal para producir la función de evaluación. Sin embargo, la función solo se usa en posiciones estáticas o quiescentes. Esto es, una posición sin situaciones tácticas como *enroques* o *jaques*. Si este no es el caso, se hace una segunda "búsqueda quiescente" (*quiescence search*) hasta que se llegue a una posición estática [Sil+17, pág. 10].

Luego se hace una búsqueda *Minimax* con poda *Alfa-beta*. Esta búsqueda además realiza una pequeña búsqueda quiescente en cada hoja [Sil+17, pág. 10]. Este es el funcionamiento básico de los motores que usan *Minimax*, *Stockfish* además usa una variedad de heurísticas para extender la búsqueda en ramas prometedoras y reducirla en ramas poco prometedoras. Algunas de estas heurísticas son: *Killer heuristic, history heuristic, Static-exchange Evaluation, Aspiration Window,* etc.

2.3.2. Leela Chess Zero

Lela Chess Zero, también conocido como *LCZero* surge como un intento de imitar a *AlphaZero* usando crowd computing [MPT22, pág. 6]. Está basado en *Leela Zero*. *Leela Zero* es un motor de *Go* hecho por Gian-Carlo Pascuatto, *LCZero* surge cuando Gary Linscott modificó este motor para que pudiera jugar ajedrez [Sil19]. *LCZero* es de código abierto, bajo la licencia *GPL*[2] y está escrito en C++.

Al igual que *AlphaZero* el motor empieza con cero conocimiento del juego además de las reglas y construye su conocimiento con el auto-juego [Som18]. Otra característica que comparte con *AlphaZero* es que usa una red neuronal para guiar un Árbol de búsqueda *Monte Carlo* [MPT22, pág. 1].

A diferencia de este *LCZero* no usa procesadores *TPU(Tensor Processing Unit*) para entrenarse sino que usa una red de computación distribuida. Esta red está formada por voluntarios que donan parte de su tiempo de procesador para ayudar a *LCZero* a entrenarse [Sil19] [Som18].

Debido a varias mejoras y entrenamiento adicional hoy podemos decir que *LCZero* ha superado por mucho a *AlphaZero* [MPT22, pág. 6].

2.4. Ajedrez Hexagonal

El primer intento de crear un juego similar al ajedrez en un tablero hexagonal fue el "Ajedrez Hexagonal" de Thomas Croughton de 1853. Para este juego se usaba un tablero de 61 casillas, las piezas eran un general, dos coroneles, dos capitanes y seis infantes [Pri07, pág. 278]. El objetivo era llevar el general al cuadrado del oponente.

Otro juego parecido era Hexagonia de John Jaques de 1864. Se jugaba en un tablero hexagonal de 127 casillas. Las piezas eran un *rey*, dos cañones, cuatro caballeros y ocho *peones*. El objetivo era ocupar la casilla central [Pri07, pág. 297]. Como podemos ver en ninguno de estos juegos el objetivo era dar *jaque mate* [Pri07, pág. 203] así que no eran realmente ajedrez.

El siguiente predecesor del ajedrez hexagonal es el juego *Mars* creado por F. H Ayres y M. van Leeuwen en 1910. Este juego representa un enfrentamiento entre Marte y la Tierra. Las casillas son hexagonales aunque

[2]La Licencia General Publica de *GNU* llamada comúnmente *GPL* es la licencia de *software* libre más conocida. Apareció por primera vez en 1989 y es esencialmente el resultado del trabajo de Richard Stallman, fundador del proyecto *GNU*. Contiene cuatro libertades esenciales: La libertad de correr el programa para cualquier propósito, la libertad de estudiar como se hizo el programa y modificarlo, la libertad de redistribuir copias del *software* y la libertad de revisar el *software* y hacer públicas las revisiones [Kum06, pág. 8].

el tablero es cuadrado. El objetivo es dar *jaque mate* (llamado "observación completa"). Las piezas son *Rey*, Sol, Luna, Astrónomo, Observatorio, Torre de Radio y Telescopio [Pri07, pág. 208].

En 1912 el ingeniero Vienés Siegmund Wellish creó una versión para tres jugadores en un tablero de 64 casillas [HW92, pág. 172]. Desde entonces se han creado muchos ajedreces hexagonales, pero el más popular es la variante creada por Wladyslaw Glinsk quien lanzó su juego en Gran Bretaña en 1949. Este fue popular en Europa oriental en el siglo XX, particularmente en Polonia, país natal del creador. Llegó a tener hasta medio millón de jugadores y en Polonia se vendieron 130,000 juegos [Pri07, pág. 203].

En 1976 se jugó el primer campeonato británico. En 1987 se jugó un campeonato mundial [HW92, pág. 172]. En la cima de su popularidad hubo una Federación Internacional de Ajedrez Hexagonal y varias organizaciones nacionales. Desafortunadamente el juego decayó después de la muerte del autor.

2.4.1. Reglas del juego

Figura 2.1: Posición inicial del tablero

Se juega en un tablero hexagonal de noventa y un casillas. Las casillas también tienen forma de hexágono, son de tres colores: claras, oscuras y de tono medio. Es un juego de dos jugadores. Cada jugador cuenta con una *dama*, dos *torres*, dos *caballos*, nueve *peones*, un *rey* y tres *alfiles*.

El *peón* se puede mover hacia el frente una casilla. Puede capturar en dirección nororiental y noroccidental. La primera vez que se mueve, se puede desplazar dos casillas. Una regla particular del ajedrez hexagonal es que si el primer movimiento de un *peón* es para capturar una pieza, y termina en la casilla inicial de otro *peón*. Entonces puede usar su segundo movimiento para mover dos casillas.

Existe la regla de captura al paso, esto es: si un *peón* usa su movimiento doble para pasar por una casilla amenazada por un *peón* contrario, el *peón* contrario lo puede capturar moviéndose a la casilla por la que pasó el primer *peón*

No hay *enroque*[3] en el ajedrez de Gliński [Pri07, pág. 204].

[3]Este término se refiere al movimiento especial de protección que permite mover al *rey* y a la *torre* en un mismo turno [She11, pág. 173].

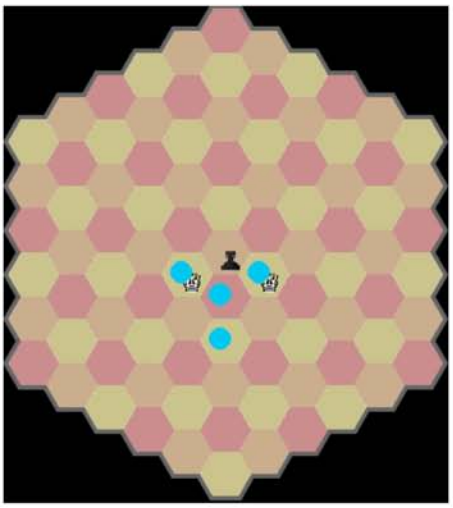

Figura 2.2: Movimientos del peón

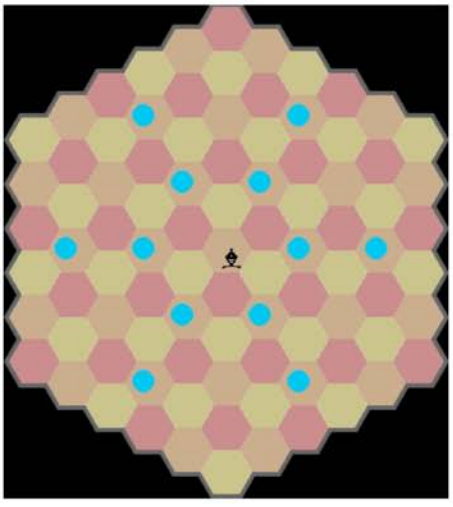

Figura 2.3: El *alfil* se puede mover sólo a casillas del color en el que empezó. Se puede mover en dirección de cada vértice del hexágono. Esto significa que se puede mover en seis direcciones.

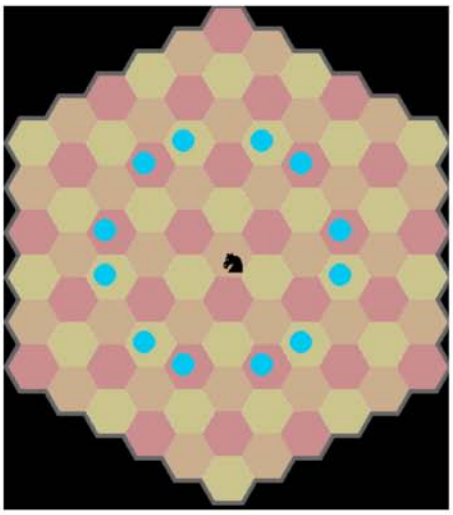

Figura 2.4: El *caballo* se mueve dando primero un "paso" de un *alfil* y luego uno de una *torre* en 30 grados. Esto forma un movimiento en "L".

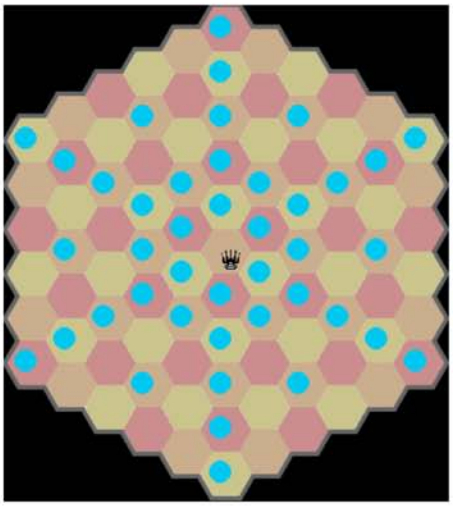

Figura 2.5: La dama combina los movimientos del *alfil* y de la *torre*.

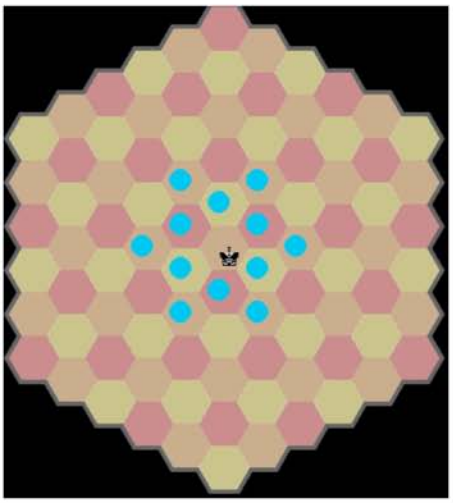

Figura 2.6: El *rey* se puede mover en la misma dirección que una dama, pero sólo una casilla.

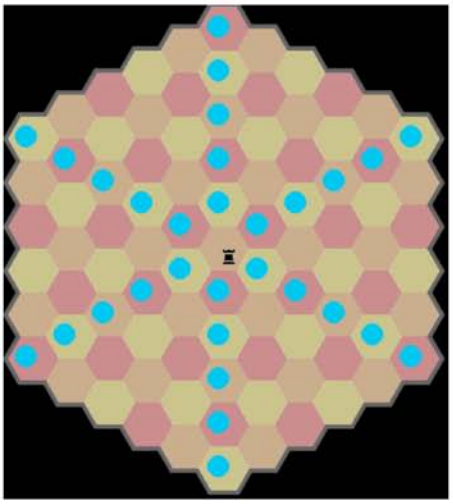

Figura 2.7: La *torre* se puede mover en línea recta en la dirección de las aristas de la casilla en la que está. Esto significa que se puede mover en 6 direcciones.

2.4.2. Implementaciones existentes del ajedrez hexagonal

A continuación se presentan las implementaciones del ajedrez hexagonal de código abierto. Solo una tiene inteligencia artificial, la mayoría sólo permiten usar el programa para que dos personas jueguen entre sí.

Autor del programa	**Características**	**URL**
codeplex	▪ Escrito en zsg. Este es un motor de juegos de mesa. ▪ Se pueden jugar tanto las variantes de McCooey como Glinski, además de otra variante con un tablero más grande. ▪ Solo es posible que dos humanos jueguen entre sí. El programa no puede jugar; solo se puede usar para jugar con otro humano.	`https://archive.codeplex.com/?p=hexchess`
abhiramavi	▪ Está escrito en *Prolog*. Tiene tres modos de juego. Puede escoger un movimiento al azar. ▪ Usa el algoritmo *Minimax* con poda *Alfa-beta*. ▪ La función de evaluación se basa únicamente en el material. ▪ No cuenta con interfaz de usuario verdadera, sino que "dibuja" el tablero con caracteres en la terminal. ▪ Es el único verdadero motor de ajedrez hexagonal de código abierto.	`https://github.com/abhiramravi/Hexagonal-Chess`
flemwad	▪ Está escrito en *Java* y es una *app* de *Android*. ▪ El programa no puede jugar; solo se puede usar para jugar con otro humano. ▪ Es una variante propia con piezas no ortodoxas.	`https://github.com/flemwad/HexChess`
seanmhanson	▪ Está escrito en *Javascript*. Funciona en el navegador. ▪ El programa no puede jugar; solo se puede usar para jugar con otro humano.	`https://github.com/seanmhanson/hexagonal-chess`

Cuadro 2.1: Programas de ajedrez hexagonal de código abierto

A continuación se presenta una lista de aplicaciones para celular de Ajedrez Hexagonal, al ser de código cerrado no se puede decir mucho de ellas.

Aplicación	**Características**
Hexagonal Chess Pass and Play	El programa no puede jugar; solo se puede usar para jugar con otro humano.
Hex-Chess	Tiene opción de jugar contra el programa, aunque su técnica es rudimentaria.
flemwad	Contiene opción de jugar contra el programa con varios niveles de dificultad.

Cuadro 2.2: Aplicaciones computacionales del ajedrez hexagonal de código cerrado

En este capítulo se revisaron tanto la historia del juego del ajedrez como la historia del ajedrez de computadora. Después se describieron brevemente los dos motores de ajedrez más notables de la época moderna. Además se presentó el ajedrez hexagonal del que se dio una breve descripción e historia. Finalmente se mencionan algunas implementaciones existentes del ajedrez hexagonal.

CAPÍTULO 3

ALGORITMOS Y TÉCNICAS

En este capítulo se describen las estructuras de datos y algoritmos usados en un motor de ajedrez típico. Primero se describe el árbol de juego, que es la estructura de datos usada para representar una partida de ajedrez. Después se describe el algoritmo *Minimax* que se usa para calcular los valores de las posiciones del tablero. A continuación pasamos a ver el algoritmo de poda *Alfa-beta*. Se mencionan brevemente dos algoritmos de poda alternativos a *Alfa-beta*. Finalmente se describe el funcionamiento de un motor de ajedrez con redes neuronales.

3.1. Árbol de juego

La estructura de datos que se usa para representar un juego de ajedrez o de juegos similares es un árbol de juego. En un árbol de juego se representan de forma explícita todas las jugadas posibles de una partida [Pea84, pág. 222]. Esta estructura fue descrita por primera vez por Claude Shannon [Abr89, pág. 137].

Podemos ver al árbol de juego como una estructura definida de forma recursiva con un nodo raíz que representa el estado actual del juego y un conjunto finito de arcos o aristas que representan los movimientos legales. Cada arista nos lleva a un subárbol de juego más pequeño [Abr89, pág. 138].

Ahora bien, se distinguen dos tipos de nodos: los nodos internos y las hojas o nodos terminales. Estos últimos no tienen ningún sucesor y representan una posición en la que ya no hay movimientos legales [Abr89, pág. 138] [Sha89, pág. 5].

Cada hoja tiene un valor numérico, este valor corresponde a una "recompensa" que, en la mayoría de los juegos, se calcula viendo si el juego se gana, se pierde o se empata [Abr89, pág. 138]. Cada camino desde la raíz hasta una hoja representa una partida completa [Pea84, pág. 222].

Es importante aclarar que al representar un juego asumimos que las reglas no permiten una secuencia infinita de posiciones y que cada estado del tablero produce solamente una cantidad finita de movimientos legales, de otra forma tendríamos un árbol infinito [KM75, pág. 294].

3.2. Factor de ramificación

Al número de arcos que salen de un nodo se le conoce como factor de ramificación. La profundidad de un nodo es la distancia entre el nodo y la raíz. Si b es el factor de ramificación promedio y d la profundidad de un árbol; entonces el árbol contiene aproximadamente $b^d + b^{d-1} + ... + b^0$ nodos. Esto se debe a que la raíz tiene un nodo $1 = b^0$. Y cada nivel agrega b^n nodos, donde n es la profundidad del nivel, hasta llegar a d. El factor de ramificación es clave para determinar la efectividad del algoritmo de búsqueda que se usará en el árbol de juego. Por ejemplo, juegos como *Starcraft* y *Civilization* tienen factores de ramificación de aproximadamente un millón, esto hace que sea impráctico enumerar las acciones, por lo que no se puede usar un árbol de búsqueda para jugarlos [YT18, págs. 103-104].

El ajedrez tiene un factor de ramificación de aproximadamente 35 [Mar90, pág. 6]. Podemos contrastar este factor con un juego como *Fox and Geese,* con un factor menor a 10. El factor de *Fox and Geese* es suficientemente bajo para que se pueda resolver exhaustivamente [Mar90, pág. 5]. Por otro lado, el juego de *Go* tiene un factor de aproximadamente 300 [YT18, pág. 45]. Esta es, posiblemente, la principal razón de que los métodos que funcionaban bien para ajedrez no funcionaran bien para *Go* [YT18, pág. 103].

En la teoría del ajedrez la partida se divide en movimientos para su análisis. Un movimiento es cuando tanto blancas como negras toman su turno y mueven una pieza. Sin embargo este término no es muy útil para el ajedrez de computadora donde se prefiere usar el "medio movimiento" que corresponde al movimiento de una pieza.

Al medio movimiento también se le conoce como "*ply*". Cada *ply* corresponde a un nivel en el árbol de juego. Consecuentemente el *ply* es la medida estándar para denotar la profundidad de una búsqueda [Abr89, pág. 144].

Un juego de ajedrez promedio tiene entre 40 y 50 movimientos. Por lo tanto, su árbol de juego, tendría de 80 a 100 *ply* [Abr89, pág. 8].

3.3. Algoritmo *Minimax*

Ya vimos cómo se representa un juego de ajedrez, sin embargo ¿cómo podemos elegir la mejor estrategia que seguir? Si queremos elegir la mejor estrategia, esta tiene que tomar en cuenta al adversario. Debe incluir un movimiento correcto para cualquier posible jugada del oponente. Esto es precisamente lo que se logra con el algoritmo *Minimax*. Ahora bien, en el ajedrez y juegos similares se alternan los turnos del jugador con los del oponente. Como el juego es de suma cero el jugador va intentar llegar al valor máximo, mientras que el

oponente tratará alcanzar el mínimo [Abr89, pág. 138]. Los jugadores se van a denominar *Max* y *Min*. *Max* es el jugador cuya recompensa queremos maximizar y Min es su oponente [RN95, pág. 124]:

1. El primer paso del algoritmo *Minimax* es generar todo el árbol del juego y expandirlo hasta que se llegue a las hojas.

2. A continuación a cada hoja se le va a dar un valor de recompensa dependiendo si se pierde, empata o gana.

3. Los nodos internos conectados a las hojas van a tomar su valor de recompensa de estas. Se escoge el nodo hijo con la mayor o menor recompensa dependiendo si es el turno de *Max* o *Min*.

4. En el segundo nivel se calcula la recompensa de los nodos internos que están a dos niveles de separación de una hoja, en este caso se sigue el mismo procedimiento, se elige al hijo con la mayor o menor recompensa.

5. Este procedimiento recursivo se continúa hasta que se llega a la raíz o estado actual. Cuando esto ocurre el algoritmo nos garantiza que llegamos a una decisión *Minimax*, esto significa que la decisión maximiza la utilidad asumiendo que el oponente va a jugar perfectamente para minimizarla [RN95, pág. 124].

3.3.1. *Minimax* acotado

Una verdadera búsqueda *Minimax* es costosa pues cada nodo hoja del árbol debe ser visitado [Mar90, pág. 6]. El problema surge en juegos con árboles de juego demasiado extensos, como el ajedrez o las damas.

Por ejemplo, se estima que un juego completo de damas tiene 10^{40} nodos [Sam, pág. 208]. Generar todo el árbol requeriría aproximadamente 10^{21} siglos, incluso si 3 billones de nodos se pudieran generar cada segundo (Perl p. 226). El tamaño de un árbol de ajedrez es todavía mayor, se estima que tiene 10^{120} nodos. Esto significa que tomaría 10^{101} siglos generarlo.

Una solución para este problema es la llamada “estrategia tipo A”. Esta estrategia consiste en terminar la búsqueda antes de llegar a las hojas, a una profundidad uniforme. Después de eso se aplica una función de evaluación heurística a los nodos en la frontera [RN95, pág. 126] y se trata este árbol acotado como si fuera completo.

3.3.2. Función de evaluación

Una función de evaluación toma una posición específica del tablero y regresa un estimado de la recompensa esperada de la partida [RN95, pág. 127]. Se usa un conjunto de características de la posición para calcular un valor numérico. Algunas de las características que se pueden usar son el material, la estructura de los *peones*, la seguridad del *rey*, etc [RN95, pág. 127].

Tener una función de evaluación de calidad es muy importante para el desempeño de un motor de ajedrez. Una mala función de evaluación puede llevar al motor a posiciones que son aparentemente buenas pero que en realidad son desastrosas [RN95, pág. 127].

¿Cómo distinguimos a una función buena? Una función de evaluación buena debe concordar con la función de recompensa en las hojas, debe tener un costo razonable y debe reflejar los chances reales de ganar [RN95, pág. 127].

Las funciones de evaluación se pueden clasificar en no lineales y lineales. En una función lineal las características se multiplican por un coeficiente y después se suman.

3.3.3. Efecto horizonte

Limitar el árbol de búsqueda significa que cualquier cosa más allá de esta frontera es invisible. Esto tiene como consecuencia un fenómeno en el que un motor de ajedrez "resuelve" una situación táctica empujando el problema fuera su campo de visión con movimientos dilatorios, esto puede ser desastroso [Mar90, pág. 13]. A este problema se le conoce como el efecto horizonte.

Una posición callada o quiescente es aquella que no se ve afectada por el efecto horizonte [Abr89, pág. 8]. Para que un motor juegue efectivamente debe poder determinar cuáles son las posiciones quiescentes y evaluar solamente estas. Este sigue siendo un problema abierto [Mar90, pág. 16][Abr89, pág. 8].

3.3.4. Complejidad del algoritmo *Minimax*

Si la profundidad máxima del árbol es m, y hay b movimientos legales en cada nivel, entonces la complejidad de tiempo es $O(b^m)$. Los requerimientos de espacio son lineares con respecto a m y b pues es esencialmente una búsqueda en profundidad modificada [RN95, pág. 126].

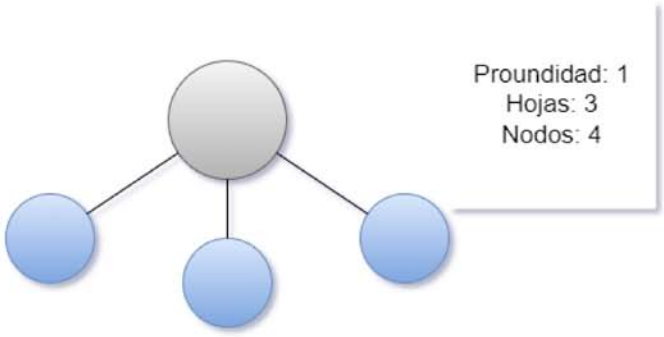

Figura 3.1: En las figuras 3.1, 3.2 y 3.3 podemos ver el crecimiento exponencial del número de hojas con respecto a la profundidad. La complejidad está dada en términos de número de hojas evaluadas. En *Minimax* simple se deben evaluar todas las hojas. Esto se puede mejorar con los algoritmos de poda. En esta figura tenemos una profundidad de 1 y tenemos 3 hojas.

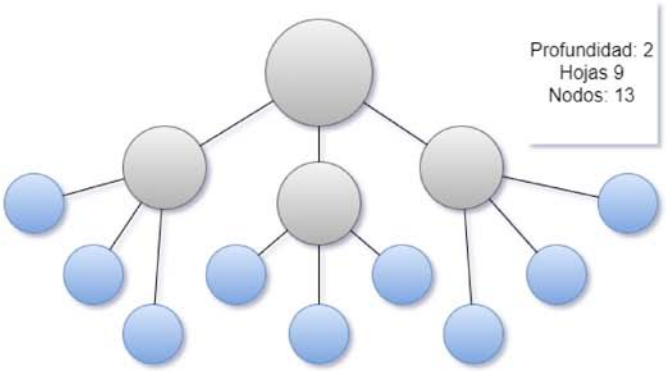

Figura 3.2: En esta figura tenemos una profundidad de 2 y 9 hojas.

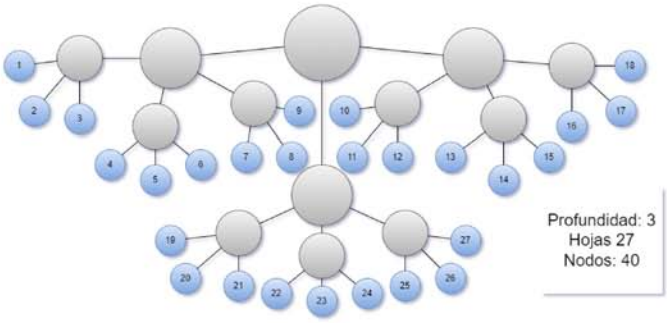

Figura 3.3: Esta es la última figura. Aquí tenemos una profundidad de 3 y 27 hojas.

3.3.5. Notación *Negmax*

La Notación *Negmax* evalúa los nodos desde el punto de vista de un sólo jugador, en vez de dos jugadores. Esto significa que se necesitan menos líneas para describir el algoritmo [Pea84, pág. 229]. Ahora veremos un ejemplo de la diferencia de notación:

```
def minimax(self):
    if self.is_leaf==True:
      self.obten_mi_valor()
      return self.valor
    else:
      lista_temp=[]
      for hijo in self.children:
        lista_temp.append(hijo.minimax())
      for hijo in lista_temp:
        if(self.soy_min==1):
          self.valor =min(lista_temp)
```

```
12            return self.valor
13          else:
14            self.valor = max(lista_temp)
15            return self.valor
```

Código 3.1: Aquí se ve una implementación recursiva de *Minimax*. Vamos a llamar la atención a la línea 10 donde se debe verificar si el nodo es *min*.

```
1  def negmax(self):
2      if self.is_leaf==True:
3        self.obten_mi_valor()
4        return self.valor
5      else:
6        lista_temp=[]
7        for hijo in self.children:
8          lista_temp.append(-hijo.negmax())
9        for hijo in lista_temp:
10         if(self.soy_min==1):
11           self.valor =max(lista_temp)
12           return self.valor
```

Código 3.2: En este código se ve una implementación de *negmax* que hace lo mismo que el código 3.1, con la diferencia de que *negmax* es más breve y no necesita verificar el tipo de nodo.

3.3.6. Ejemplo de uso de algoritmo *Minimax*

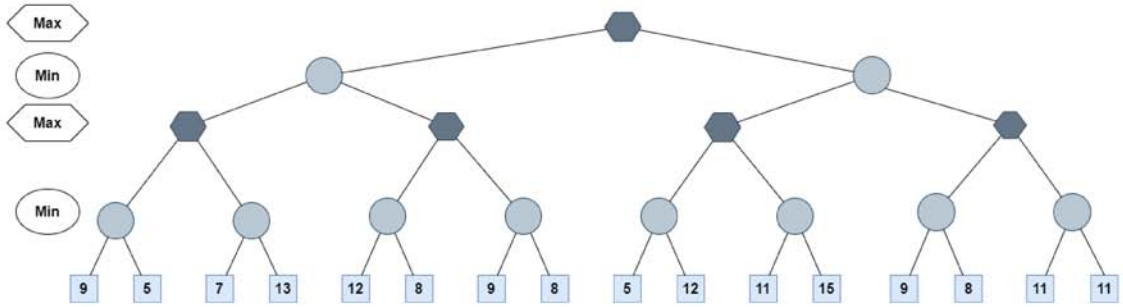

Figura 3.4: Ejemplo de árbol de decisión. En este momento solo tiene el valor de las hojas. Primero se explora el árbol hasta una profundidad uniforme. Se usa la función de evaluación para determinar el valor de los nodos en la frontera.

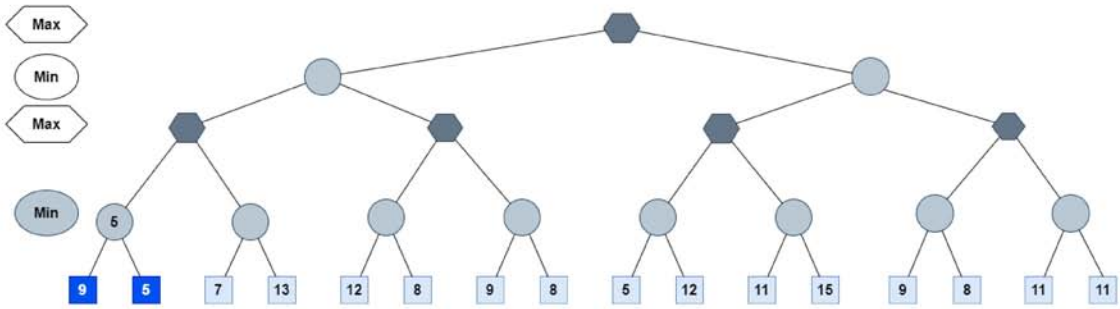

Figura 3.5: En esta figura el árbol ya tiene el valor de un nodo interno calculado. El primer nodo interno es *Min*, por lo tanto su valor va a ser el valor del menor hijo, en este caso 5.

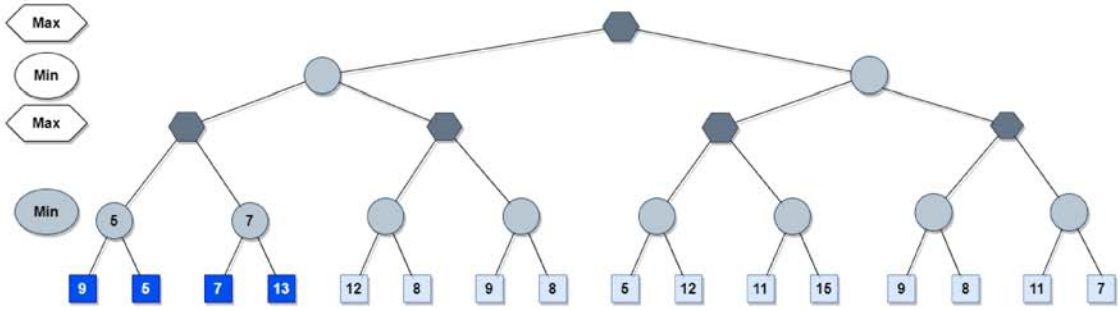

Figura 3.6: Ahora haremos lo mismo para su nodo hermano. Este va a ser el segundo nodo calculado

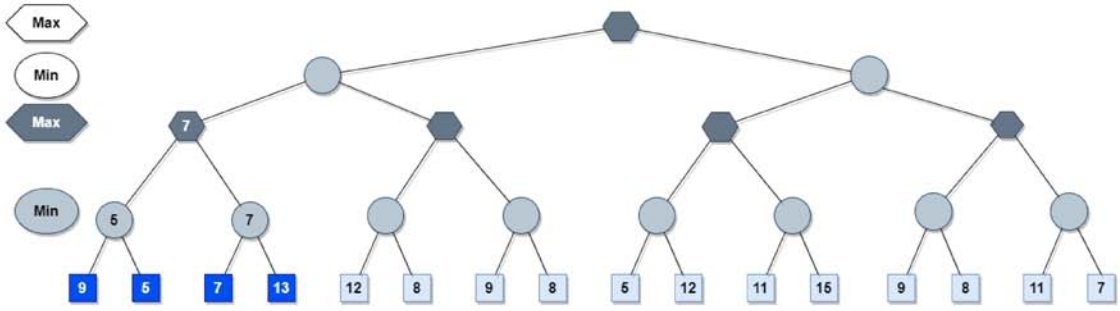

Figura 3.7: El siguiente nodo es *Max*, por lo tanto se elige el valor correspondiente al hijo con mayor valor. Este es el tercer nodo.

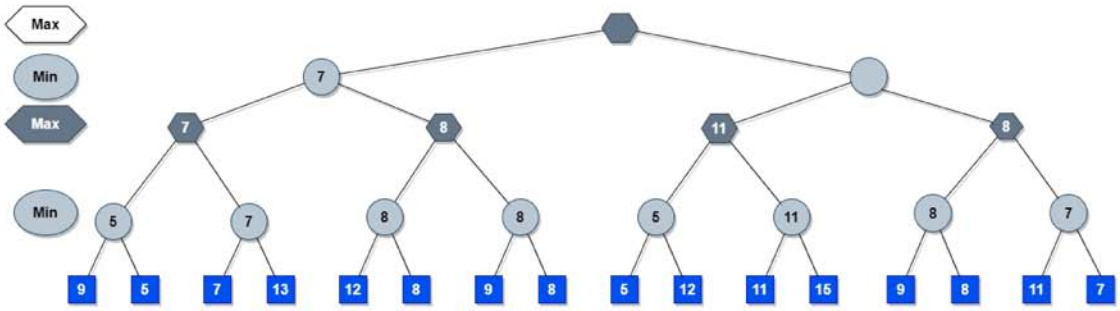

Figura 3.8: El proceso se continúa de forma recursiva. En esta imagen ya están calculados la mayoría de los nodos.

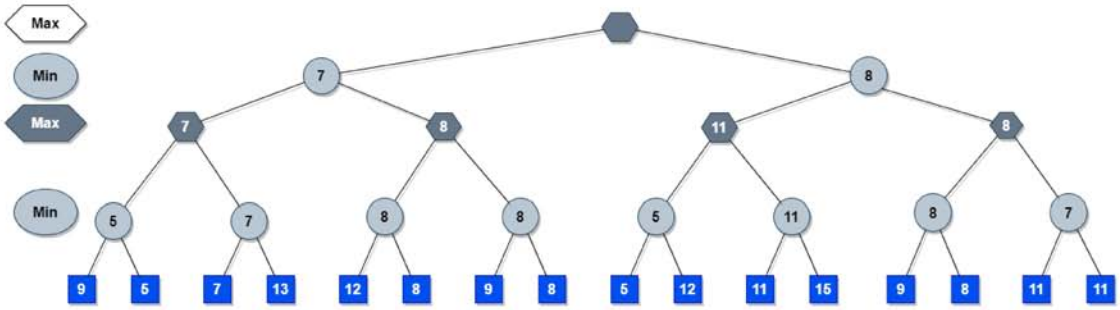

Figura 3.9: Sólo falta un nodo por calcular

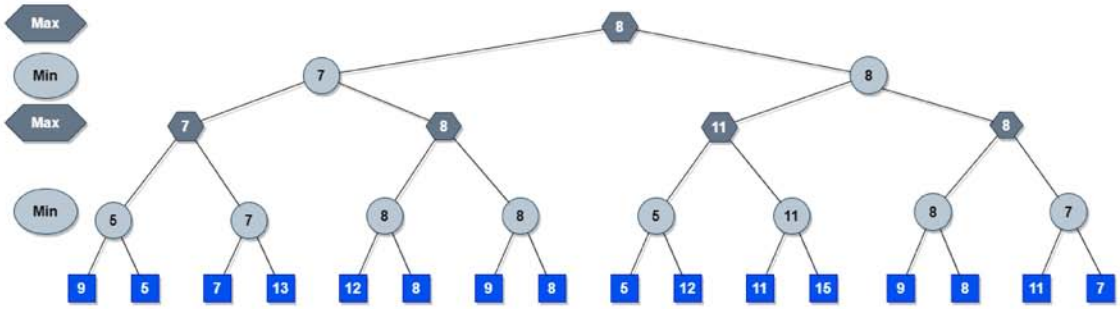

Figura 3.10: En esta figura ya está resuelto el árbol. Al llegar a la raíz obtenemos que el mejor valor es 8. Por lo tanto, la decisión *Minimax* es seguir el camino del hijo derecho.

3.4. Algoritmo de poda *Alfa-beta*

Ahora bien, como hemos visto un motor tiene dos tareas de cómputo importantes, generar el árbol de juego y calcular el valor de una posición con la función de evaluación. Es importante encontrar un balance entre

estas dos actividades y evitar cómputos innecesarios. La forma más significativa de evitar estos cómputos innecesarios es evaluar solamente los nodos prometedores e interesantes y no desperdiciar el tiempo en posiciones malas. El algoritmo de poda *Alfa-beta* nos garantiza que va a obtener el valor *Minimax* del árbol saltando todos los nodos que no contienen información relevante [Pea82, pág. 560].

Hay que aclarar que a pesar de su efectividad *Alfa-beta* no elimina el crecimiento exponencial del árbol de juego [Sha89, pág. 1204].

3.4.1. Historia de *Alfa-beta*

La primera persona en tener la idea fundamental del algoritmo de poda *Alfa-beta* fue John McCarthy, uno de los padres fundadores de la IA. Tuvo la idea durante la Conferencia de Investigación sobre Inteligencia Artificial de Verano de Dartmouth en 1956. Bernstein estaba dando una conferencia explicando el funcionamiento de *Minimax* cuando McCarthy criticó al algoritmo, proponiendo un algoritmo de poda para mejorarlo. Sin embargo Bernstein no fue convencido y McCarthy no escribió una especificación formal del algoritmo [KM75, pág. 303].

Alfa-beta es utilizado en el famoso programa de Damas de Arthur Samuel de 1959. Pero Samuel no dio un especificación formal del algoritmo ni le prestó atención pues le parecía evidente y poco importante, en vez de eso su artículo se enfoca en la función de evaluación que desarrolló [Her18, pág. 165].

La primera discusión publicada de un método para la poda de un árbol de juego es la descripción hecha en 1958 por Alan Newell, John Cliford Shaw y Herbert Simon de uno de sus programas de ajedrez tempranos. Sin embargo, no es claro si usaron cortes profundos pues solamente describen la técnica de un solo lado [KM75, pág. 303] [Her18, pág. 164].

El algoritmo fue descrito de forma detallada por primera vez en 1963 en un artículo en la publicación rusa "Problemas de la cibernética" escrito por Alexander Brudno. Este desarrollo ocurrió independientemente del trabajo en América [KM75, pág. 303] [Mar90, págs. 6, 32].

La técnica de poda *Alfa-beta* por ambos lados apareció en occidente en 1968, en un artículo titulado *Experiments With a Multipurpose, Theorem-Proving Heuristic Program*(Experimentos con un programa heurístico multipropósito para probar teoremas) escrito por James Slagle y Philip [SB68, págs. 92-93] [KM75, pág. 304]; sin embargo la descripción es muy breve y no muestra cortes profundos [KM75, pág. 304]. Se puede decir que, en el mundo occidental, no hubo una descripción detallada del algoritmo que mostrará cortes profundos sino hasta el año de 1969. Año en el que esta descripción apareció en dos artículos diferentes, uno por Slagle y Dixon; y otro por Arthur Samuel [KM75, pág. 304].

En un inicio se creía que *Alfa-beta* era una heurística y no un algoritmo verdadero hasta que en 1975 Donald Knuth y Ronald Moore probaron que la poda *Alfa-beta* siempre produce el mismo resultado que el algoritmo *Minimax* [Her18, pág. 167].

3.4.2. Descripción del algoritmo *Alfa-beta*

El algoritmo *Alfa-beta* funciona registrando dos límites entre los cuales debería estar el valor del nodo [Abr89, pág. 140]. Estos límites nos servirán para registrar información sobre los nodos ya visitados, esta información la usaremos para podar ramas que no se van a visitar.

- **Límite alfa**: Supongamos que estamos considerando un nodo tipo *Min* llamado J. Alfa va a corresponder al valor actual más alto de todos los ancestros tipo *Max* de J. Podemos hacer un corte en J si su valor es igual o menor a alfa [Pea84, pág. 234].

- **Límite beta**: Supongamos que estamos considerando un nodo tipo *Max* llamado K. Beta va a corresponder al valor actual más bajo de todos los ancestros tipo *Min* de K. Podemos hacer un corte en K si su valor es igual o mayor a beta [Pea84, pág. 234].

Es importante notar que los límites Alfa y Beta se inicializan en menos infinito e infinito [Abr89, pág. 140]. Hay un ejemplo del funcionamiento de la poda *Alfa-beta* en la sección 3.5.4

3.4.3. Complejidad *Alfa-beta*

Los primeros en analizar a profundidad la efectividad de *Alfa-beta* fueron Donald Knuth y Ronald Moore en su artículo de 1975. Encontraron que cuando los sucesores están ordenados aleatoriamente la complejidad asintótica es $O((\frac{b}{logb})^d)$ donde b es el factor de ramificación promedio y d la profundidad [RN95, pág. 132].

3.4.4. Ejemplo de uso de algoritmo de poda *Alfa-beta*

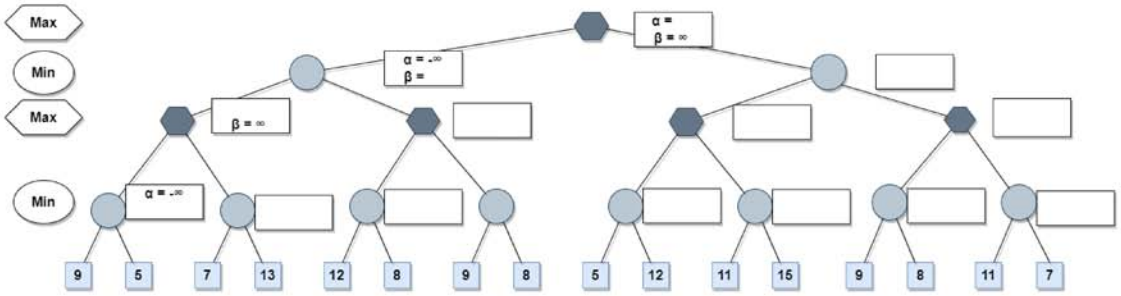

Figura 3.11: Se usa el mismo árbol que en el ejemplo *Minimax*. Los valores iniciales son $\alpha = -\infty$ y $\beta = \infty$.

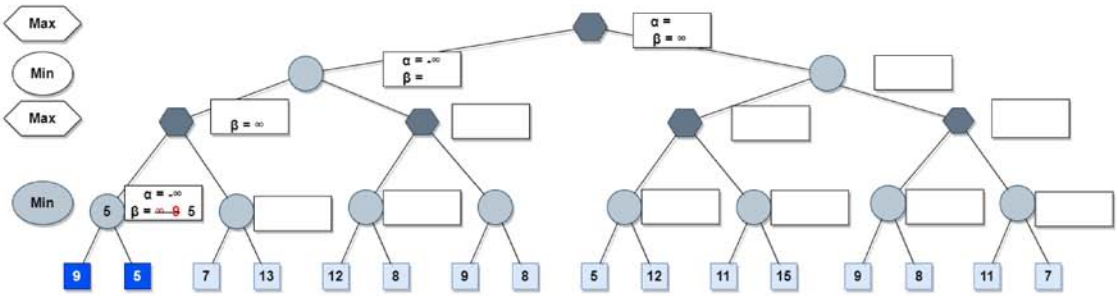

Figura 3.12: El primer nodo es *Min* por lo tanto sólo se actualiza su valor β.

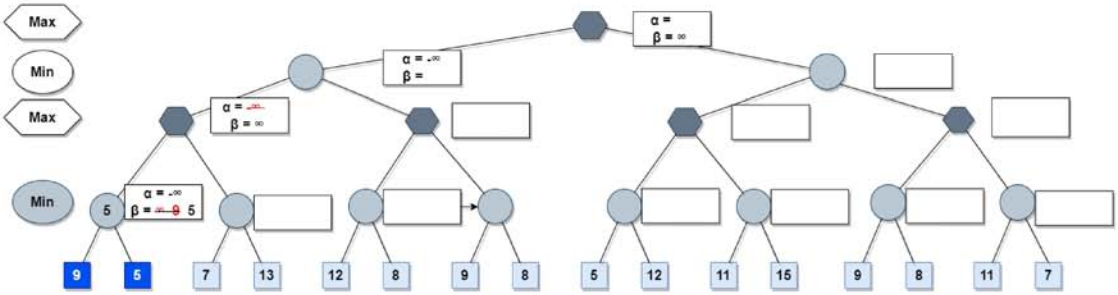

Figura 3.13: El padre se actualiza con el máximo valor de los hijos. En este caso 5, al ser un nodo *Max*, sólo se actualiza α.

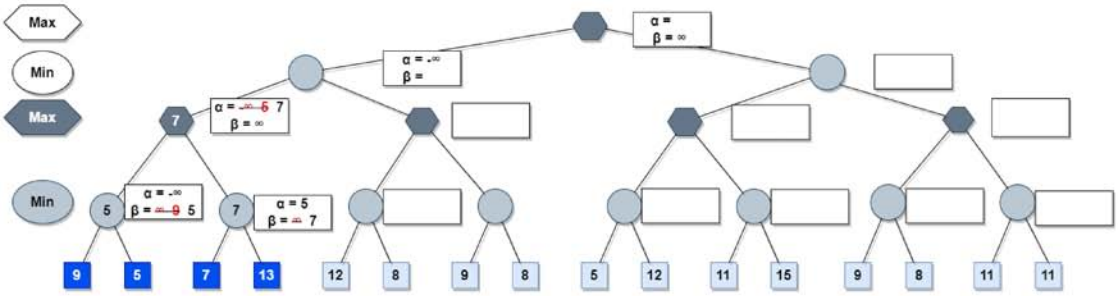

Figura 3.14: Ahora se repite el proceso para el nodo hermano y se vuelve a actualizar el padre.

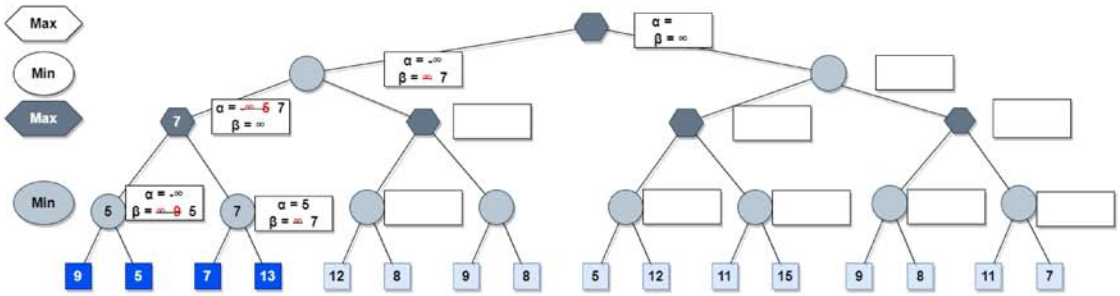

Figura 3.15: Ahora que se completó el nivel *Max* se puede actualizar el abuelo, esto es, el siguiente nivel *Min*.

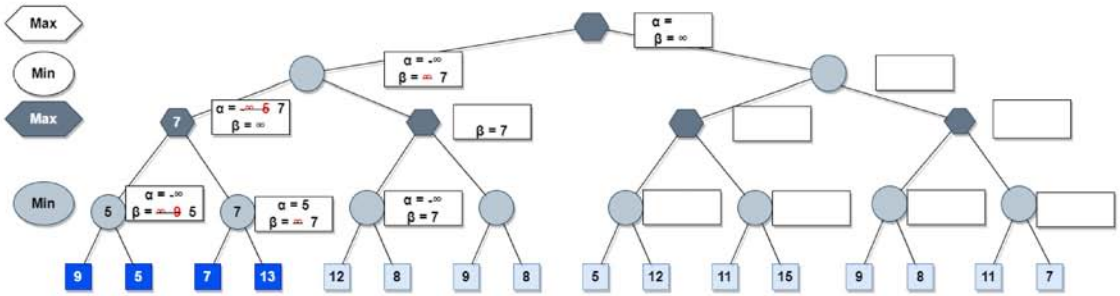

Figura 3.16: El nuevo valor se hereda al subárbol derecho.

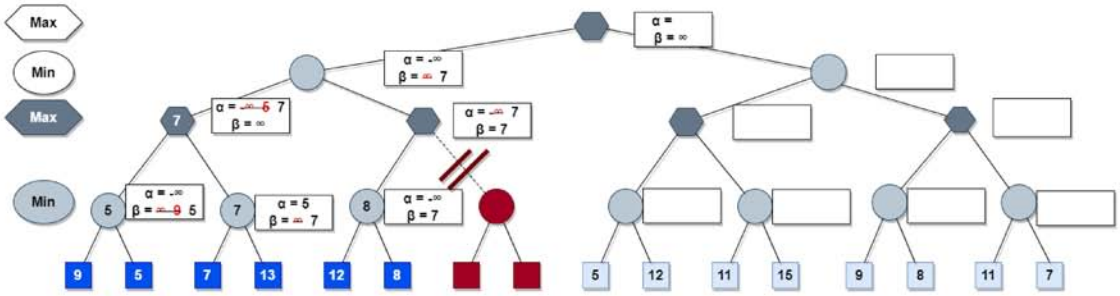

Figura 3.17: Ahora se repite el proceso. Aquí llegamos a un punto de poda. Como $\alpha = \beta$ podemos podar la rama derecha.

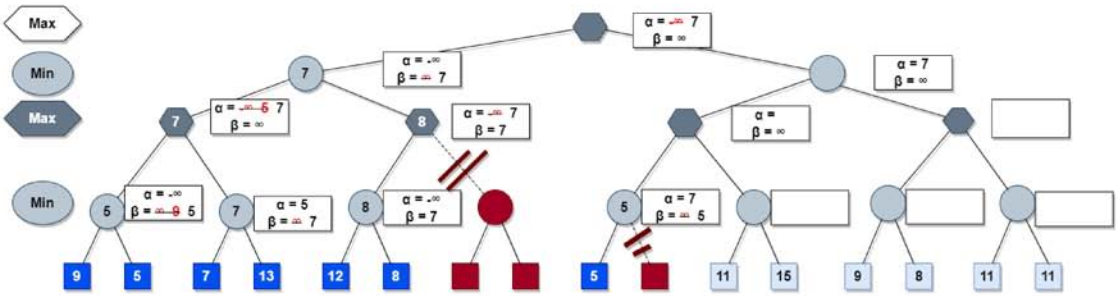

Figura 3.18: Continuamos el proceso recursivamente hasta el siguiente punto de poda. En este caso se puede podar porque α es mayor a β.

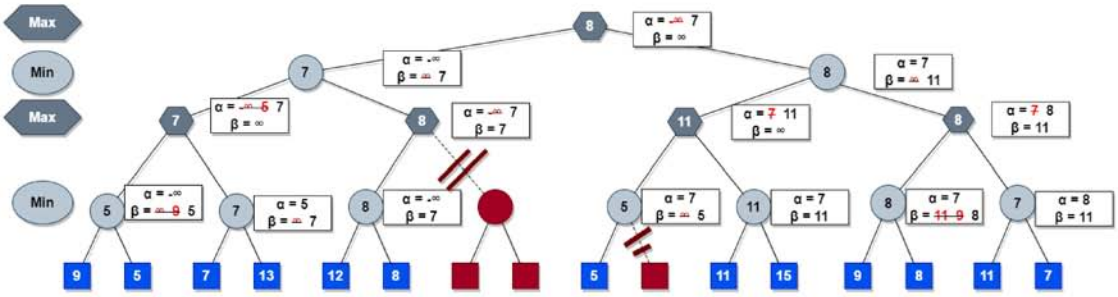

Figura 3.19: El proceso se continúa recursivamente hasta terminar y llegar a una decisión *Minimax*.

3.5. Algoritmos alternativos

3.5.1. SSS*

Este algoritmo fue introducido en 1979 por George Stockman [Abr89, pág. 140]. Originalmente fue desarrollado para resolver problemas de análisis de la forma de ondas [Pea84, pág. 286].

Funciona recorriendo subárboles usando un método de primero el mejor, similar al usado por A* [1]. Para guardar los subárboles utiliza una lista llamada *OPEN*. Esta lista es de tamaño exponencial con respecto a la profundidad del árbol.

Se ha demostrado que SSS* domina a *Alfa-beta* pues nunca evalúa un nodo cortado por *Alfa-beta* y ocasionalmente evalúa menos nodos [Pea84, pág. 240]. Su desventaja es que la lista *OPEN* ocupa un espacio demasiado grande, además de que debe estar ordenada [Pla+14, pág. 5].

[1] A* Es un algoritmo cuyo propósito es encontrar el camino más corto entre dos nodos en una gráfica, es similar al algoritmo de Dijkstra. Usa una política de "mejor primero", una función heurística y una lista para guardar los caminos posibles. (Zeng p.9)

3.5.2. Scout

Este algoritmo fue desarrollado por Judea Pearl en 1980 [Abr89, pág. 141]. Curiosamente el autor tenía la expectativa de que iba a ser dominado por *Alfa-beta* cuando creó este algoritmo [Pea84, pág. 249]. Pero resultó que supera a *Alfa-beta* en ciertas situaciones [Pea84, pág. 250].

Este algoritmo funciona haciendo una "prueba" (*test*) rápida a los sucesores de un nodo. Esta prueba es menos costosa que la evaluación completa. Dependiendo del resultado de la prueba se decide si se va a hacer una evaluación completa del nodo o no [Pea84, pág. 246][Abr89, pág. 141].

A diferencia de SSS*, este algoritmo no tiene demandas excesivas de espacio [Pea84, pág. 250]. Esto ha permitido que sea realmente útil para el desarrollo de de juegos [Abr89, pág. 141].

Marsland y Campbell desarrollaron el algoritmo *Principal Variation Search*(Búsqueda de variación principal) como una variante del algoritmo Scout optimizada para el cómputo paralelo [Abr89, pág. 141].

3.6. Motores con redes neuronales

En los últimos años ha habido un auge de motores de ajedrez que usan redes neuronales [HH21, pág. 6]. Algunos ejemplos exitosos son *AlphaZero* [Woo21] y *Leela Chess Zero* [MPT22, pág. 1]. A continuación describiremos algunos de los elementos usados en estos.

3.6.1. Redes Neuronales

Una red neuronal artificial, mejor conocida por sus siglas en inglés: *ANN (Artificial Neural Network)* es un modelo computacional inspirado en las redes de neuronas biológicas [Pur+16, pág. 12]. Consiste en un conjunto de neuronas artificiales interconectadas, cada una de estas toma un conjunto de valores de entrada y produce una salida después de realizar una operación matemática sencilla.

El primer modelo de una neurona artificial fue elaborado en 1943 por el neurólogo Warren McCulloch y el matemático Walter Pitts quienes construyeron una red neuronal primitiva usando circuitos eléctricos llamado modelo *MCP* [WR17, pág. 9]. Este modelo podía tomar un conjunto de entradas y producir un valor de 1 o 0 dependiendo de una función de umbral. *MCP* tiene una limitación importante: carece de método de aprendizaje.

Un personaje fundamental para el desarrollo de las redes neuronales fue el neurólogo Donald Hebb, aunque él no trabajó directamente en las redes neuronales su trabajo fue muy influyente en el campo. Especialmente la llamada "Regla de aprendizaje de hebbiana" que apareció en su libro *The Organization of Behaviour* publicado en 1949 [WR17, pág. 7]. Esta ley explica que cuando una conexión entre dos neuronas se usa mucho, esta conexión se fortalece y se hace más eficiente.

La investigación de Hebb así como el trabajo de McCulloch y Pitt inspiraron al psicólogo Frank Rosenblatt a crear el Perceptrón [Nil, pág. 92]. El perceptrón toma un conjunto de entradas binarias, las multiplica por un

peso continuo, las suma y finalmente usa una función umbral para producir un 1 o 0 como salida [Kur20]. El mecanismo de aprendizaje es el cambio de los pesos de las aristas.

El perceptrón se implementó como *Hardware* en el Laboratorio de Aeronáutica de la universidad de Cornell donde se usó para clasificar figuras. Este perceptrón tomaba como entradas imágenes de 20x20 pixeles [Kur20].

Los perceptrones tienen la capacidad de aprender cualquier cosa que puedan representar. Sin embargo, son pocas las cosas que pueden representar, esto fue descubierto por Marvin Minsky y Seymour Papert, quienes publicaron este descubrimiento en su libro *Perceptrons* en 1969 [RN95, pág. 21]. Este descubrimiento disminuyó el entusiasmo por las redes neuronales y también su financiación.

A pesar de la falta de interés que hubo por las redes neuronales por varios años, gradualmente se lograron resolver las limitaciones de los perceptrones usando redes neuronales multicapa así como el algoritmo de *Backpropagation* (propagación hacia atrás).

En una red neuronal multicapa existen capas ocultas que no interactúan ni con la entrada ni con la salida sino que toman sus entradas y salidas de otras capas [YT18, pág. 61].

El algoritmo de *Backpropagation* fue creado en 1969 por Byron y Ho [RN95, pág. 21], este algoritmo sirve para corregir los pesos de una red neuronal comparando el resultado que produce con el resultado esperado. En la década de los 80s se convirtió en el estándar de facto para entrenar redes neuronales multicapa, permitiéndoles abordar problemas más complejos. Durante este tiempo, el trabajo pionero de investigadores como Geoffrey Hinton, Yann LeCun, entre otros, sentó las bases para la futura explosión de las redes neuronales.

Otra innovación importante en el uso de las redes Neuronales ocurrió cuando se comenzaron a usar *GPUs* para entrenar las redes Neuronales [Kur20]. Y es que los *CPUs* que se usaban tradicionalmente tienen una capacidad de paralelismo limitada. El poder del entrenamiento con *GPUs* fue demostrado por Abde-Rahman Mohamed, George Dahl y Geoff Hinton con su trabajo en reconocimiento de voz [Kur20].

El verdadero potencial de las redes neuronales comenzó a ser evidente con la llegada del "*Deep Learning*" a principios del siglo XXI. Al agregar más capas a las redes neuronales artificiales, se pudo lograr un nivel de abstracción y aprendizaje previamente inalcanzable.

3.6.2. *TD-Gammon*

El primer gran éxito de las redes neuronales en el área de los juegos fue en el *Backgammon*. En 1992 Gerald Tesauro desarrolló *TD-Gammon* [YT18, pág. 8] que usando el auto-juego alcanzó a jugar al mismo nivel que un maestro. *TD-Gammon* usaba un Perceptrón Multi-Capa (*Multi-Layer Perceptron*) . Su entrada era el estado actual del tablero y la salida la probabilidad de victoria [YT18, pág. 86].

Para entrenar la red neuronal Tesauro puso al programa a jugar contra sí mismo . Después de jugar 300,000 juegos alcanzó el mismo nivel de los mejores programas de *Backgammon* del momento. Esto fue un hito, pues los programas tradicionales se basan en usar conocimiento experto mientras que *TD-Gammon* fue construido con cero conocimiento [SB98, pág. 488].

3.6.3. *AlphaGo*

AlphaGo fue un motor de *Go* elaborado por *Deepmind*, usaba dos redes neuronales. Una red de valor (value network) que servía para evaluar una posición de tablero y una red de política (policy network) que sirve para elegir el mejor movimiento. La red de política se entrenaba en dos fases. Primero con aprendizaje supervisado usando una base de datos de partidas. En una segunda fase se le entrenan usando aprendizaje por refuerzo usando auto-juego. Las redes se combinaban con el algoritmo *MCTS* (*Monte Carlo Tree Search*) [Woo21, pág. 89].

En 2016 venció al campeón de *Go*, Lee Sedol en Corea del Sur. Se jugaron cinco juegos de los que *AlphaGo* ganó cuatro y perdió uno [Woo21, pág. 89]. Para entrenarse usaba procesadores especiales llamados *TPU* .

3.6.4. *AlphaGo* Zero

A diferencia de *AlphaGo* esta versión usa solamente aprendizaje por refuerzo. Empieza con cero conocimiento de *Go* además de las reglas y subsecuentemente usa el autojuego para entrenar [Woo21].

3.6.5. AlphaZero

Este motor fue elaborado también por *DeepMind* como una generalización aún más avanzada de *AlphaGo Zero* . Tiene la capacidad de jugar *Go*, Ajedrez y *Shogi* [Sil+17, pág. 1]. Al igual que su predecesor sólo necesita que se le programen las reglas del juego y después puede entrenarse sólo jugando consigo mismo.

En el caso del ajedrez se le dieron cuatro horas para entrenarse, usando Unidades de Procesamiento de Tensor, del inglés *Tensor Processing Unit*, *TPU*. Después de su entrenamiento pudo vencer consistentemente a *Stockfish* [Sil+17, pág. 4]. Además pudo vencer al motor de *Shogi Elmo*, que era campeón mundial.

3.6.6. Redes Convolucionales

Existen varios tipos de redes neuronales como las redes generativas antagónicas, las redes neuronales pre alimentadas o las redes neuronales recurrentes. Pero la más relevante para el ajedrez de computadora han sido las llamadas redes neuronales convolucionales.

La convolución es un tipo especializado de operación lineal. Las redes neuronales convolucionales emplean la convolución en lugar de la multiplicación general de matrices en al menos una de sus capas [GBC16, pág. 326].

3.6.7. Árbol de Búsqueda *Monte Carlo*

El árbol de búsqueda *Monte Carlo* es un algoritmo que construye un árbol de Juego y luego realiza una búsqueda en él. A diferencia de *Minimax*, se basa en la estadística, y no en una función de evaluación

heurística [Ewa12, pág. 16]. Fue creado por Rémi Coulom en el 2006.

La idea distintiva del *MCTS* es la simulación *Monte Carlo*. Esta consiste en tomar la posición del nodo actual y jugar muchas partidas hasta el final, los movimientos se eligen de forma aleatoria. Después de jugar un cierto número de partidas, le podemos asignar un valor al nodo dependiendo de los resultados. Por ejemplo, si las negras ganaron muchas veces desde esta posición esto se va a ver reflejado en este valor. Como podemos ver, con el *MCTS* no es estrictamente necesaria una función de evaluación [Sil09, pág. 5]. El algoritmo tiene 4 fases: descenso, expansión, despliegue y propagación hacia atrás [Ewa12, págs. 16-17].

- **Descenso:** En está etapa se debe escoger un nodo qué expandir. Para esto se siguen dos criterios:
  - Se busca expandir nodos que no han sido explorados aún.
  - Se busca expandir los nodos más prometedores, esto está determinado por el valor de cada nodo

  ¿Cómo se le asigna valor a los nodos? Hay distintas formas de determinarlo, se pueden usar redes neuronales o fórmulas. La forma más estándar es usar la fórmula *UCT* [Ewa12, pág. 18]. Está forma toma en cuenta el valor de los hijos, el número de veces que se ha visitado el nodo, etc.
- **Expansión:** Se expande el nodo elegido generando un nodo hijo.
- **Despliegue o Simulación:** Se hace una simulación de un juego basada en el nodo del paso anterior. Se realizan jugadas aleatorias hasta que termine el juego [Ewa12, pág. 17].
- **Propagación hacia atrás:** El resultado del despliegue se propaga hacia los ancestros del nodo. Un ejemplo es que se tiene que modificar el número de “victorias” de sus nodos ancestros.

Este proceso se continúa hasta que se cumpla alguna condición arbitraria de salida [Ewa12, pág. 17]. A la hora de elegir el nodo podemos usar uno de varios criterios: más victorias, más visitas, mejor razón de visitas a pérdidas, etc.

3.6.8. Aprendizaje por refuerzo

Aprendizaje por refuerzo: En el aprendizaje por refuerzo el agente vive en un entorno del cual recibe estímulos constantes y con el que puede interactuar. El agente toma decisiones que afectan su ambiente y este le provee retroalimentación o refuerzo.[en] la forma de una recompensa o un castigo. Con el paso del tiempo estos estímulos producen un aprendizaje en el agente y de esta forma se construye una "política"que maximiza el desempeño del agente [Lin92, pág. 294].

Los juegos de tablero son un ambiente ideal para este paradigma, pues el perder, empatar o ganar una partida es una función natural de refuerzo.

El tema principal de este capítulo fue el funcionamiento de un motor de ajedrez. Primero se explican algunos de los conceptos más importantes del mundo de los motores de ajedrez, como son el árbol de juego, el factor de ramificación y el *ply*. Después pasamos a describir el algoritmo *Minmax*, el más importante en la

historia del ajedrez de computadoras. Proseguimos a describir el algoritmo de poda *Alfa-beta* y se mencionan muy brevemente algunos algoritmos alternativos. En lo que resta del capítulo se habla de otra familia de motores de ajedrez. Esta familia usa redes neuronales y el algoritmo del árbol de búsqueda *Monte Carlo*.

CAPÍTULO 4

DESCRIPCIÓN DEL PROGRAMA

En este capítulo se hablará más detalladamente del programa que se realizó. Empezamos dando una breve descripción de la metodología y tecnologías que se usaron, estos son los lenguajes de programación y las bibliotecas gráficas. Después pasamos a la descripción del programa.

Esta descripción consiste en tres partes. La primera parte es una descripción de cómo se tradujo el problema al paradigma orientado a objetos. La segunda parte consiste en describir los principales problemas que se tuvieron que resolver y cómo se resolvieron.

La última parte consiste en una explicación de la interfaz de usuario. Finalmente incluimos un *benchmark* en el que se comparan los dos lenguajes de programación que se usaron: *Python* y C++ así cómo los dos métodos que se usaron: Algoritmo *Minimax* simple y Algoritmo *Minimax* con poda *Alfa-beta*.

4.1. Metodología de desarrollo

En el desarrollo del motor de Ajedrez se intentó usar la metodología *Crystal Clear*. Las metodologías *Crystal* surgieron como resultado del trabajo que Alistar Cockburn hizo para *IBM* en la década de los 90s [Coc04, pág. 29]. *Crystal Orange* fue descrita en el libro *Surviving Object-Oriented Project*s de 1998.

Las metodologías *Crystal* fueron precursoras del manifiesto *Agile*, del que Cockburn fue signatario y siguen los principios de esta declaración [Bec+]:

- Individuos e interacciones sobre procesos y herramientas
- *Software* funcionando sobre documentación extensiva
- Colaboración con el cliente sobre negociación contractual

- Respuesta ante el cambio sobre seguir un plan

Crystal es una familia de metodologías, están clasificadas por colores dependiendo del tamaño y el peso del proyecto. La metodología relevante para nuestro proyecto es *Crystal Clear* para equipos pequeños. El objetivo de *Crystal Clear* es generar *software* satisfactorio priorizando la eficiencia, la habitabilidad y la seguridad [Coc04, pág. 531].

Todas las metodologías *Crystal* tienen en común siete principios:

- **Entregas frecuentes:** se espera que las entregas sean semanales o quincenales. La idea es que se puedan encontrar problemas de manera temprana.
- **Mejora reflexiva:** esto significa que el equipo interrumpe el trabajo intermitentemente para discutir el proceso, analizarlo y si es necesario cambiarlo.
- **Comunicación cercana u osmótica:** esto involucra que todos los miembros del equipo estén enterados de lo que hacen sus compañeros.
- **Seguridad personal:** se debe promover un ambiente respetuoso en el que todos los miembros del equipo puedan expresar sus opiniones y críticas.
- **Enfoque:** esto se refiere a que un equipo se debe enfocar a una sola tarea el tiempo suficiente para que se haga progreso.
- **Acceso fácil para usuarios expertos:** se debe intentar que haya un usuario experto que no sea parte del equipo de desarrollo.
- **Ambiente técnico:** se busca que haya pruebas automáticas, administración de configuraciones e integración frecuente. La idea es que se puedan detectar y corregir errores rápidamente.

4.2. Tecnologías utilizadas para el desarrollo de *software*

4.2.1. Lenguaje de programación C++

C++ es un lenguaje de programación compilado, con tipificado estático, y sin recolección de basura. Soporta tanto programación orientada a objetos como estructurada. Fue creado por Bjarne Stroustrup basándose en el lenguaje C. En esta sección la información es tomada principalmente del libro *Concepts of Programming Languages* de Robert W. Sebesta.

El objetivo de Stroustrup al crear C++ era combinar las habilidades de organización de Simula con la eficiencia y el poder de C. Esta idea surgió mientras trabajaba en su tesis doctoral en Cambridge [Str, pág. 1].

Stroustrup comenzó a diseñar el lenguaje en 1979 mientras trabajaba en los laboratorios Bell en Nueva Jersey [Str, pág. 4]. Las modificaciones de C iniciales incluyeron la adición de clases, acceso público y privado

a los componentes heredados, métodos constructores, métodos destructores y clases amigas [Seb96, pág. 88]. El lenguaje resultante fue llamado C con clases. Una descripción temprana de este lenguaje fue publicada en un reporte técnico en abril de 1980 [Str, pág. 5].

La compatibilidad con C fue una prioridad desde el inicio. Pensando en esto no se removió casi ninguna característica de C; ni siquiera las que eran consideradas inseguras [Seb96, pág. 88].

En 1984 se incluyeron las funciones virtuales, estas fueron la característica más controversial e incomprendida del lenguaje [Str86, págs. 208-210]. También en este año se renombró el lenguaje a C++ [Str86, pág. 208]. En octubre de 1985 aparece la primera versión comercial de C++.

En la segunda mitad de la década de los 80 se continuaron agregando funcionalidades basándose en los comentarios de los usuarios, entre estas funcionalidades estuvieron las plantillas (templates) y el manejo de excepciones. Una funcionalidad que causó controversia en su tiempo fue la herencia múltiple, agregada en 1987. Años después la herencia múltiple se volvió estándar en los lenguajes orientados a objetos con tipado estático [Str86, pág. 211].

Para 1987 estaba claro que una estandarización formal de C++ era inevitable, así que se decidió emprender el largo camino hacia el estándar [Str86, pág. 212]. El primer paso fue elaborar el manual de referencia que serviría como base para este proyecto, esto se hizo con la ayuda de implementadores de compiladores y usuarios. El comité X3J16 de *ANSI (American National Standards Institute)* fue convenido en 1989 por iniciativa de *Hewlett-Packard*. Más tarde también se formó un comité *ISO (International Standards Organization)*. Stroustrup sirvió en ambos comités. Finalmente en 1998 se ratificó el estándar *ISO* 14882 [Str86, pág. 212]. Desde entonces han habido varias revisiones y actualizaciones del estándar.

En el año de 1998 se agregó el marco de trabajo (*framework*) *STL* con varios algoritmos y contenedores a la biblioteca estándar. Este fue el trabajo de Alex Stepanov con la ayuda de Dave Musser, Meng Lee y otros [Str86, pág. 211].

En la actualidad C++ es uno de los lenguajes más populares. Un factor en su popularidad es la disponibilidad de compiladores buenos y de bajo costo [Seb96, pág. 89]. Otro factor importante es su eficiencia en tiempo de ejecución, y es que C++ fue pensado para usarse en sistemas y nunca se ha sacrificado su desempeño [Str, pág. 41]. También hay que reconocer que apareció en el momento adecuado, pues a finales de los 80s e inicios de los 90s era el único lenguaje orientado a objetos con la capacidad de ser usado en proyectos grandes [Seb96, pág. 89].

El sistema de tipos de C++ es estático. El tipificado estático es una técnica en los lenguajes de programación en la que, durante el proceso de compilación, el verificador de tipos intenta asignar a los objetos su tipo particular. La verificación estática significa que las expresiones y variables a las que se les asignó sus valores son revisadas para asegurarse de su exactitud antes de que el programa sea ejecutado.

Se usó C++ en la segunda versión del programa pues se buscaba mejorar la velocidad de este así como el uso de recursos.

4.2.2. Lenguaje de programación *Python*

Python es un lenguaje de programación interpretado, con recolector de basura y tipado dinámico. Soporta Programación Orientada a Objeto, programación estructurada, y también contiene algunos elementos de programación funcional. Fue diseñado a inicios de los 90s por el computólogo neerlandés Guido van Rossum cuando trabajaba en el *Stichting Mathematisch Centrum* en Países Bajos [Seb96, pág. 99]. El lenguaje está inspirado en un lenguaje anterior llamado ABC y fue desarrollado originalmente para el sistema *Amoeba* [Gui]. Actualmente su desarrollo está a cargo de la *Python Software Foundation* [Seb96, pág. 99].

Python usa tipificado dinámico, esto significa que los tipos son revisados solamente durante la ejecución del programa, a diferencia del tipificado estático que los revisa en tiempo de compilación [HS15, pág. 22].

4.2.3. Capa Simple de Medios Directos

La Capa Simple de Medios Directos traducido del inglés *Simple Direct Media Layer*, mejor conocido por las siglas *SDL*.

SDL es una biblioteca multimedia para C y C++, fue creada por Sam Lantinga en 1997 cuando trabajaba en *Loki Software* [Yuz06, pág. 6] y está disponible en múltiples plataformas, entre ellas *Windows, OS X* y *Linux* [Mit13, pág. 6].

SDL provee al programador de una interfaz que brinda acceso de bajo nivel a los componentes de video, audio y a métodos de entradas [Mit13, pág. 5] [Yuz06, pág. 6].

Algunos de los juegos exitosos desarrollados con esta biblioteca son: *World of Goo, Neverwinter Nights* y *Second Life*. También se ha usado en emuladores como *Zsnes, Mupen65* y *VisualBoyAdvance* [Mit13, pág. 5].

4.2.4. Pygame

Pygame es una biblioteca perteneciente al marco de trabajo de *Python* y su propósito es ayudar en la creación de videojuegos. Fue creada por Pete Shinners y utiliza *SDL* [Idr13, pág. 7]. *Pygame* es gratis y de código abierto desde 2004 y está bajo la licencia *GPL*. Se decidió usar esta tecnología porque es la forma más directa y sencilla de crear la interfaz de usuario.

En nuestro motor *Pygame* se usa para mostrar las imágenes en pantalla, para mostrar los diálogos al usuario y para cambiar la forma del cursor.

4.3. Descripción del programa de juego de Ajedrez

Ahora se va a describir el programa, primero se hace una breve descripción de las clases que se usaron para el modelado del ajedrez y luego se describen los principales problemas que se tuvieron que resolver durante la creación del programa.

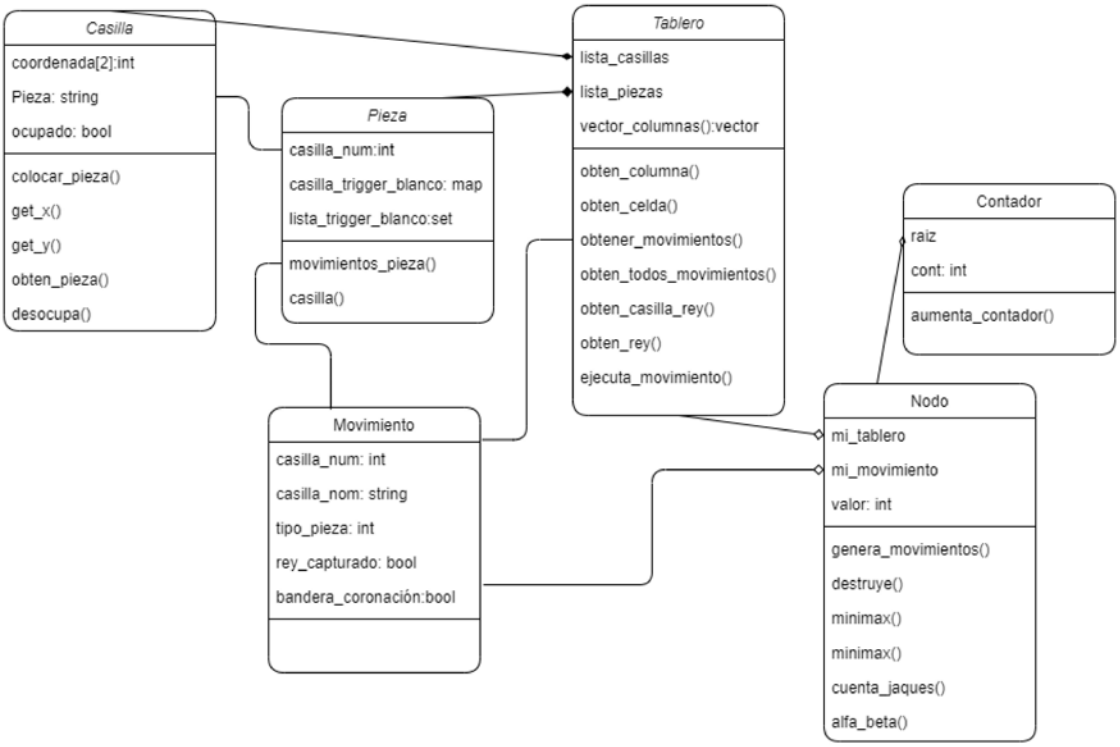

Figura 4.1: Diagrama de clases del programa.

4.3.1. Modelado de las partes del juego

En esta sección se hablará del modelado del programa. Veremos qué clases se usaron y para qué sirve cada clase.

- **Tablero:** el tablero consiste en el conjunto de casillas y piezas. Cada una con un nombre único que se usa para acceder a ellas.

- **Casilla:** la casilla guarda la información que indica si está libre o ocupada, así como la pieza que la ocupa. Cada casilla está asociada a varias columnas y filas, estas sirven para determinar los movimientos de los *peones, reyes, alfiles, damas* o *torres* que se coloquen en ella.

- **Piezas:** en la clase pieza están contenidos los métodos para obtener los movimientos de cada tipo de pieza. Por ejemplo, que una *torre* se puede mover por las columnas, pero solo hasta encontrarse con otra pieza, si esta pieza es enemiga la puede capturar, pero si es amiga no.

- **Movimiento**: consiste en una pieza, que es la que se va a mover y una casilla de destino.

- **Nodo:** un objeto tipo nodo contiene un tablero y una lista de movimientos. El tablero es hipotético, indica la posición de las piezas si se hace un movimiento dado. Cada nodo hijo tiene un "tablero hipotético" que corresponde a cómo se vería el tablero si se hicieran las de sus padres. Si construimos un camino desde el nodo raíz hasta una hoja este camino nos muestra una partida posible entre muchas. La lista de movimientos indica los movimientos posibles desde la posición del tablero correspondiente al nodo. Además hay un diccionario que indica las casillas a las que se puede mover un *caballo* colocado en la casilla.

4.3.2. Problemas a resolver para desarrollar el programa

Se tuvieron que resolver 5 problemas esenciales: dibujar el tablero, mover las piezas en el tablero, determinar los movimientos posibles dado un tablero, encontrar un movimiento en respuesta al del usuario y determinar el fin de la partida.

I. Dibujar el tablero

Este problema se resuelve poniendo una imagen de un tablero en la ventana, las piezas se dibujan visitando cada casilla, si la casilla está ocupada se toma la imagen de la pieza correspondiente y se dibuja en las coordenadas que corresponden a la casilla.

II. Movimiento de piezas

En este problema es dónde se usa la interfaz de usuario. Para mover una pieza se usa la posición del mouse, esta posición se empata con las coordenadas de cada casilla. De esta manera sabemos qué pieza quiere mover el jugador y a dónde la quiere mover. Se deben generar los movimientos posibles de la pieza tomada por el usuario, para asegurarse de que el movimiento que realizó el usuario sea válido. Además aquí hay que considerar el problema del *jaque*. Cuando un *rey* está en *jaque* sólo se deben permitir los movimientos que ponen al *rey* fuera de peligro, así que antes de permitir al jugador mover una pieza debemos revisar dos cosas: que sea un movimiento válido y que no ponga al *rey* en *jaque*, si el *rey* ya se encuentra en *jaque* el movimiento lo debe sacar de él.

III. Determinar los movimientos posibles

Este problema se resuelve con la clase pieza y la clase casilla. La casilla contiene las columnas, filas, diagonales, etc. en las que se encuentra la casilla. La clase pieza contiene los métodos que dictan cómo se mueve cada pieza.

En el caso de la captura al paso, este movimiento especial requiere recordar qué pieza se movió en el anterior turno y hacia dónde.

El otro caso especial es el de la promoción, este movimiento no sólo mueve una pieza sino que esta tiene la capacidad de convertirse en una variedad de piezas, cada una de las opciones se debe considerar como un movimiento válido.

IV. Determinar el movimiento que la computadora va a jugar

Para determinar el mejor movimiento para la máquina se usa el algoritmo *Minimax* con poda *Alfa-beta*. Esta técnica requiere una función de evaluación.

La función de evaluación nos da un valor dependiendo si un tablero es bueno para el jugador o malo. En este caso usamos dos funciones de evaluación. Una para el *endgame* y otra para el resto de la partida.

La función del *endgame* busca poner al *rey* oponente en *jaque*. La función normal toma en cuenta solamente el valor del material. A cada pieza se le da un valor dependiendo de su valor aproximado, por ejemplo, las piezas más valiosas, excluyendo al *rey*, son la *dama* y la *torre*. La menos valiosa es el *peón*.

En cada turno de la máquina se construye un árbol de búsqueda que tiene como raíz la situación actual del tablero. Cada nodo del árbol contiene un tablero hipotético y cada arista representa un movimiento.

Para generar el árbol tenemos un número límite de nodos, el árbol se construye usando una búsqueda por anchura(*Bread-first search*). Para hacer esto se usa una estructura *heap*.

Después de que se construyó el árbol, que siempre tiene un número n de nodos se pasa a aplicar la función de evaluación a las hojas.

Después de eso se usa el algoritmo *Minimax* con poda *Alfa-beta* para determinar el mejor movimiento para la máquina

V. Determinar el fin de la partida de ajedrez

En cada tablero se debe determinar si es un tablero terminal, esto es si se llegó a una condición de final de la partida. Hay dos formas de terminar la partida:

- Empate: si el jugador en turno no tiene ningún movimiento válido, pero su *rey* no está en *jaque* entonces la partida se considera un empate
- *Jaque Mate*: Si el jugador en turno no tiene ningún movimiento válido y su *rey* está en *jaque* entonces se considera que perdió.

Ambas condiciones se revisan con el mismo método, primero revisa si existen movimientos válidos y luego si está en *jaque*, dependiendo de la combinación de estos dos factores la partida continúa o termina en empate o victoria. El programa no tiene en cuenta el empate por movimientos repetidos o el empate por número de movimientos.

4.3.3. Extractos de código

En esta sección vamos a explicar el código de algunas secciones representativas del código.

```
int Nodo::alfa_beta(){
  if (this->is_leaf == true){
    obten_mi_valor();
    this->nombre = to_string(this->indice) + "val " + to_string(this->valor) + "
    alfa " + to_string(this->alfa) + " beta " + to_string(this->beta);
    return this->valor;
  }
  else{
    if (this->soy_min == true){
      for (auto& hijo : hijos){
        hijo->alfa = this->alfa;
        hijo->beta = this->beta;

        int valor_hijo = hijo->alfa_beta();
        if (valor_hijo < this->beta){
          this->valor = valor_hijo;
          this->beta = this->valor;
        }
        if (this->alfa >= this->beta){
          this->nombre = to_string(this->indice) + "val " + to_string(this->valor) +
     " alfa " + to_string(this->alfa) + " beta " + to_string(this->beta);
          return this->valor;
        }
      }
      this->nombre = to_string(this->indice) + "val " + to_string(this->valor) + "
    alfa " + to_string(this->alfa) + " beta " + to_string(this->beta);
      return this->valor;
    }
    else  {
      for (auto& hijo : hijos){
        hijo->alfa = this->alfa;
        hijo->beta = this->beta;
        int valor_hijo = hijo->alfa_beta();
        if (valor_hijo > this->alfa)  {
          this->valor = valor_hijo;
          this->alfa = this->valor;
        }
        if (this->alfa >= this->beta){
          this->nombre = to_string(this->indice) + "val " + to_string(this->valor) +
     " alfa " + to_string(this->alfa) + " beta " + to_string(this->beta);
          return this->valor;
        }
      }
      this->nombre = to_string(this->indice) + "val " + to_string(this->valor) + "
    alfa " + to_string(this->alfa) + " beta " + to_string(this->beta);
      return this->valor;
```

```
42      }
43    }
44  }
```

Código 4.1: Algoritmo *Alfa-Beta* del archivo Nodo.cpp

Lo primero que vamos a notar es que el método es de tipo entero. Este valor de retorno corresponde al valor de la función heurística del mejor hijo. En la línea 2 se revisa si el nodo es hoja. De resultar hoja se llama a la función `obten_mi_valor()`. Esta función va a su vez a llamar a una función heurística para obtener el valor del tablero. La línea 4 simplemente nombra al nodo y la 5 regresa el valor del nodo, este valor se actualizó en la línea 3.

En la línea 8 se prueba si el nodo es *Min*. En las líneas 10 y 11 se heredan los valores de alfa y beta, estos se inicializaron en el constructor del Nodo a -10000 y 10000.

La variable `valor_hijo` de la línea 3 es una variable temporal para guardar el resultado de la recursión.

En la línea 14 se hace una prueba, si el valor del hijo es menor a beta entonces se actualiza el valor de beta y también el valor del nodo.

En la línea 18 es donde tenemos la condición de corte. Si el valor alfa es mayor o igual al valor beta entonces para el proceso recursivo, ya no evalúes al resto de los hijos.

Las líneas 19 y 23 se aseguran de que el nombre del nodo siempre sea actualizado. La línea 24 se ejecuta si no hubo ningún corte.

En las líneas 27 a 44 vemos el caso de que el nodo sea *Max*. Es análogo así no que no hay mucho que decir de ellas que no se haya dicho del caso *Min*.

```
1
2  void dibuja_tablero(Tablero miTab, SDL_Surface* mi_superficie, bool
       dibuja_movimientos){
3    SDL_FillRect(mi_superficie, NULL, SDL_MapRGB(mi_superficie->format, 0xFF, 0xFF, 0
       xFF));
4    SDL_Rect stretchRect;
5
6    stretchRect.x = 0;
7    stretchRect.y = 0;
8    stretchRect.w = ALTURA_TABLERO;
9    stretchRect.h = ANCHURA_TABLERO;
10
11   SDL_BlitScaled(gTablero, NULL, mi_superficie, &stretchRect);
12   dibuja_circulos(mi_superficie, miTab);
13   int turno = miTab.turno;
14   dibuja_turno(mi_superficie, turno);
15
16   if (dibuja_movimientos){
17     dibujar_movimientos(miTab, mi_superficie);
18   }
```

```
19   SDL_UpdateWindowSurface(gWindow);
20 }
```

Código 4.2: función dibuja_tablero del archivo funciones_dibujo.cpp

Ahora veremos la función `dibuja_tablero`. Esta función controla todo lo que se dibuja en la pantalla.

Los argumentos son un objeto tablero, una superficie SDL, este es el canvas donde se va a dibujar y un booleano.

Lo primero que se hace es rellenar la ventana con un color negro, esto se hace en la línea 3.

Las líneas 6 a 9 ajustan el tamaño de la ventana de acuerdo a las constantes definidas en el archivo `Constantes.h`

En la línea 11 se pone la imagen `gTablero` en la ventana, esta imagen se declaró al inicio del archivo y se cargó en `loadMedia()`

Después se llama a la función `dibuja_circulos`, esta función se va a encargar de dibujar las piezas.

En la línea 13 se obtiene el turno del tablero, esto se usará en la línea 14 para dibujar el marcador del turno.

En la línea 16 tenemos una condición. `dibuja_movimientos` nos va a indicar si debemos dibujar los movimientos posibles. Esto ocurre cuando el usuario está .ªgarrandoüna pieza

Finalmente en la línea 19 se actualiza la ventana.

```
1  def movimientos_alfil(self):
2    movimientos_posibles = []
3
4    mi_fila =  self.casilla().fila_alfil
5    self.agrega_movimentos_lista_retrocede(self.casilla(), mi_fila,
       movimientos_posibles)
6    self.agrega_movimentos_lista_avanza(self.casilla(), mi_fila, movimientos_posibles)
7    mi_diagonal_arriba =  self.casilla().diagonal_alfil_arriba
8    self.agrega_movimentos_lista_retrocede(self.casilla(), mi_diagonal_arriba,
       movimientos_posibles)
9    self.agrega_movimentos_lista_avanza(self.casilla(), mi_diagonal_arriba,
       movimientos_posibles)
10
11   mi_diagonal_abajo =  self.casilla().diagonal_alfil_abajo
12   self.agrega_movimentos_lista_retrocede(self.casilla(), mi_diagonal_abajo,
       movimientos_posibles)
13   self.agrega_movimentos_lista_avanza(self.casilla(), mi_diagonal_abajo,
       movimientos_posibles)
14
15   return movimientos_posibles
```

Código 4.3: función movimientos_alfil del archivo clase_pieza.py

Las funciones de movimientos funcionan basándose en una casilla. Por ejemplo este método regresará una lista de casillas a las que se podría mover un alfil que se encuentre en una casilla determinada

`movimientos_posibles` esta es la lista donde se guardan las casillas. En la línea 4 vamos a tomar una lista de casillas y llamarla `mi_fila`. Esta fila de casillas la obtenemos de la propia casilla asociada a la pieza,

En la línea 5 se usa el método `movimientos_lista_retrocede`. Esté método recibe tres argumentos:

1. una casilla, esta debe ser elemento del segundo argumento
2. una lista de casillas, de aquí se van a tomar casillas que se agregarán al tercer argumento
3. una segunda lista de casillas: Este es lo único que se modifica, se le van agregando casillas.

Se van a agregar las casillas hasta que se tope con una pieza o se acabe la "fila".

El método en la línea 6 hace exactamente lo mismo pero en dirección opuesta.

Este mismo procedimiento se repite otras 4 veces: 2 por cada sentido por las otras 2 direcciones en las que se mueve el alfil.

De esta manera toma 6 operaciones los movimientos del alfil pues se mueve en 3 direcciones

4.3.4. interfaz de usuario

Después de iniciar, el programa nos muestra un menú de opciones. Podemos elegir cuál versión queremos jugar: La que fue hecha en C++ o la que fue hecha en *Python*, también podemos elegir si queremos jugar contra la IA o no. Podemos elegir si queremos usar la versión con *Minimax* simple o con poda *Alfa-beta*, finalmente nos ofrece la opción de poner el número de nodos del árbol de juego.

Figura 4.2: menú inicial

Después el programa es preguntar al usuario, a través de la terminal. Si quiere jugar con las piezas blancas o negras a través de un cuadro de diálogo con dos botones (ver figura 4.3)

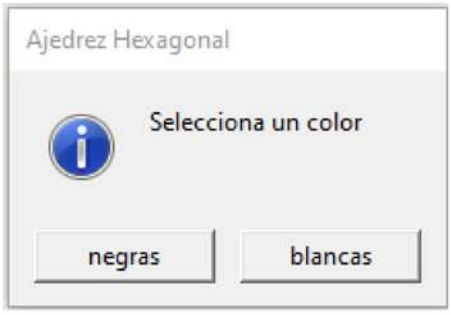

Figura 4.3: Menú de color de las piezas

La interfaz es muy intuitiva, las piezas se toman y se colocan en la casilla en la que se quieran colocar. El círculo en la esquina inferior derecha indica de qué jugador es el turno. Por ejemplo en la figura 4.4 se observa éste, del lado inferior izquierdo y como está en color blanco denota que es el turno de las piezas blancas.

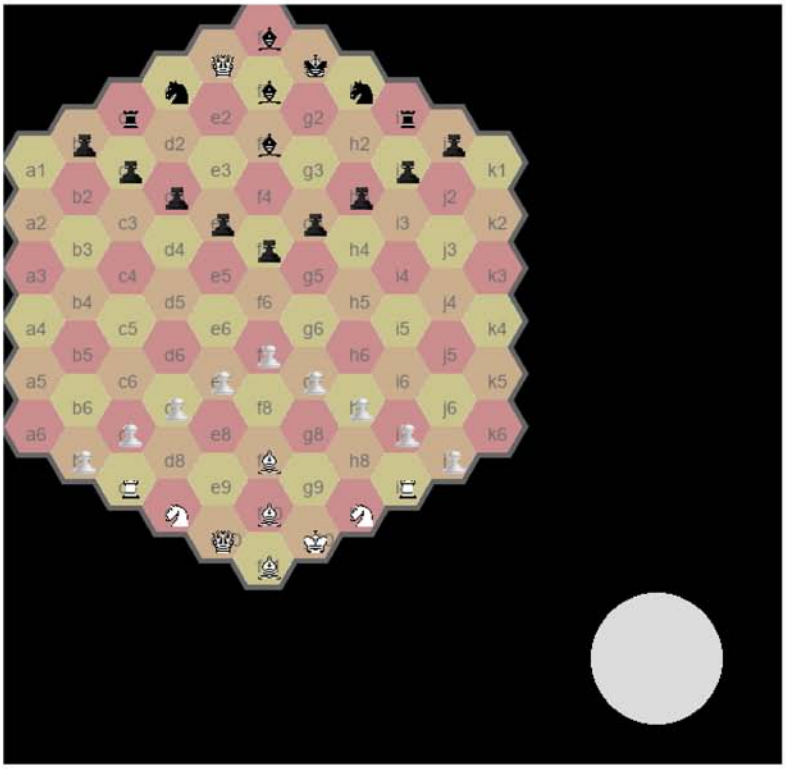

Figura 4.4: Interfaz del juego

El juego dibuja un círculo de color rojo sobre las casillas a las que se puede mover una pieza.

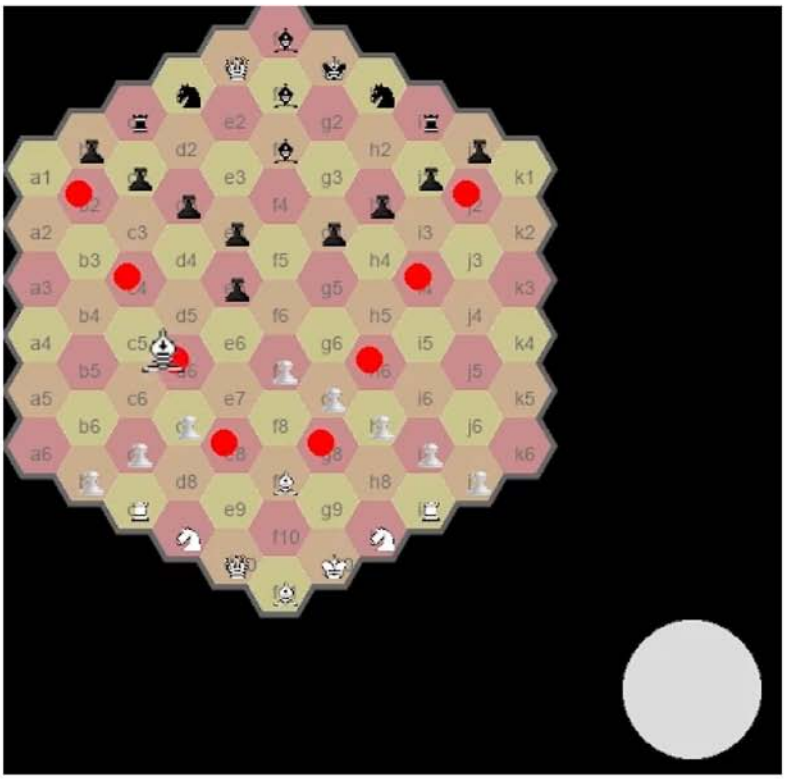

Figura 4.5: Las casillas se colorean con los movimientos disponibles. En este caso se tomó el *alfil* colocado en la casilla *f10*, se colorean de rojo las casillas a donde se puede mover el *alfil*, si se pusiera en una casilla distinta el *alfil* regresaría a su posición original en *f10*.

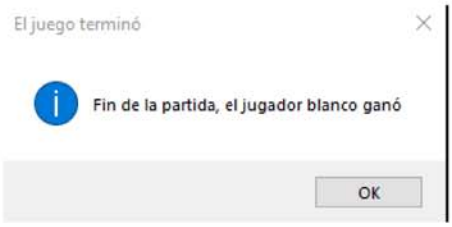

Figura 4.6: Final de la partida

Si se intenta mover a una casilla que no está dentro de los movimientos posibles para la pieza en cuestión, el programa ignorará el movimiento. Si el *rey* del jugador está en *jaque*, el programa forzará al jugador a elegir una jugada que lo ponga fuera de peligro. Ya sea moviendo al *rey* a una casilla segura, capturando a la pieza que está realizando el *jaque* o bloqueando el *jaque*. Cuando un *peón* llega a una casilla de promoción, el programa le pregunta al usuario a cuál pieza lo quiere promocionar, esto se hace mediante un cuadro de diálogo con cuatro botones.

Cuando el juego llega a una condición final aparece un mensaje que indica que la partida finalizó. Además indica qué jugador ganó o si fue un empate.

Cuando la partida finaliza, también finaliza la ejecución del programa. Para jugar otra vez se tiene que volver a iniciar el programa.

4.4. *Benchmark*

Se usó la herramienta *hyperfine* para realizar el *benchmark* de las distintas versiones. En las siguientes tablas se presentan los resultados después de ejecutar cada programa 10 veces. Cada vez el programa evaluó un árbol de 3000 nodos. Se usó la misma computadora para todos.

4.4.1. *Python Minimax*

Tiempo (promedio $\pm$ σ)	14.129s $\pm$ 0.093s
Rango (min…max)	14.040….14.342 s

Cuadro 4.1: *Python Minimax*. σ da la desviación estándar, el tiempo está dando segundos.

4.4.2. *Python Minimax* con poda *Alfa-beta*

Tiempo (promedio $\pm$ σ)	14.132s $\pm$ 0.122s
Rango (min…max)	13.962 ….14.346 s

Cuadro 4.2: *Python Minimax* con poda *Alfa-beta*

4.4.3. C++ *Minimax*

Tiempo (promedio $\pm$ σ)	3.600s $\pm$ 0.264s
Rango (min…max)	3.364 ….4.145 s

Cuadro 4.3: C++ *Minimax*

4.4.4. C++ *Minimax* con poda *Alfa-beta*

Tiempo (promedio $\pm$ σ)	3.690s $\pm$ 0.817s
Rango (min…max)	3.268 ….5.952 s

Cuadro 4.4: C++ *Minimax* con poda *Alfa-beta*

Como podemos ver en *Python* no hay una diferencia significativa entre el programa *Minimax* simple y *Minimax* con poda *Alfa-beta*. Por otro lado, en C++ contrario a lo esperado el programa con *Minimax* simples es más rápido que el que usa poda *Alfa-beta*.

En este capítulo se describieron brevemente las tecnologías usadas en el programa. Es importante hablar de ellas porque son los cimientos donde se construye la estructura del programa. Estas tecnologías son los lenguajes de programación C++, *Python* y sus respectivas bibliotecas de juegos: *Pygame* y *SDL* Después

se dio una descripción del programa enfocándose en 3 aspectos: El modelado, los problemas a resolver y la interfaz de usuario. Esto es relevante pues el programa de ajedrez hexagonal es el centro de esta tesis.

Finalmente damos los resultados del *benchmark* de rendimiento que se hizo con las distintas versiones del programa. La implementación que se hizo es importante pues es un paso para la expansión de la inteligencia artificial hacia un juego poco explorado. Además examinamos qué tan efectivas son dos técnicas clásicas del ajedrez de computadora.

CAPÍTULO 5

CONCLUSIONES

El objetivo principal de la tesis era crear un programa de ajedrez hexagonal eficiente y retador para un jugador humano de nivel principiante. Este objetivo se cumplió:

- Podemos decir que el programa es eficiente en su versión de C++, pues C++ usa menos memoria que lenguajes como *Perl*, *Python* y *Java* además de tener un menor tiempo de ejecución [Pre00, pág. 29]. Esto se debe a varios factores, uno importante está asociado a que utiliza un compilador en los diversos pasos que se requieren para generar código ejecutable, además de que la eficiencia ha sido uno de los objetivos de diseño de C++ desde sus inicios. Otra de sus ventajas es que todo lo que se escribe en este lenguaje, usando las primitivas del lenguajes se asegura que no generan un "sobrecosto" en relación al desempeño y uso de recursos, pues algunas de sus primitivas se encuentran implementadas bajo el lenguaje C. Es en este sentido importante que un motor de ajedrez sea eficiente pues el crecimiento exponencial del árbol de juego hace que cualquier ahorro de recursos sea indispensable.
- El motor de ajedrez hexagonal desarrollado juega de manera efectiva en el medio juego [1] (*midgame*), aunque sufre de debilidades en las fases de apertura y en el final. Estas debilidades se deben a la falta de una tabla de aperturas y de una mejor función de evaluación elaborada con conocimiento experto. Sin embargo estas debilidades no impiden que el juego sea retador e interesante para un jugador principiante.

Ahora analizaremos cuáles de los objetivos secundarios se lograron:

- El primer objetivo secundario planteado al inicio de este trabajo es el poder estudiar este tipo de juegos implementados a través de una herramienta dentro del área de inteligencia artificial aplicada a una

[1] El medio juego es la segunda etapa de una partida de ajedrez. Comienza cuando ambos jugadores han desarrollado sus piezas menores y han resguardado a su *rey*. Termina cuando quedan pocas piezas en el tablero, aunque no hay una línea clara entre el medio juego y el final.

variante del ajedrez. Podemos decir que este objetivo se cumplió pues se estudiaron la efectividad de dos técnicas y dos lenguajes de programación.

- El segundo objetivo secundario era analizar dos formas distintas de hacer un motor de ajedrez. Este objetivo se cumplió parcialmente, se analizaron dos técnicas: *Minimax* con y sin poda *Alfa-beta*, pero los resultados obtenidos fueron contrarios a los esperados, esto probablemente se debe a la función de evaluación que es demasiado sencilla. Otra área de oportunidad para desarrollar una siguiente fase de este programa sería el profundizar en el estudio con distintas funciones de evaluación.
- El tercer objetivo secundario era crear una interfaz de usuario intuitiva para el juego. Este objetivo se cumplió pues la interfaz de usuario funciona muy bien y es fácil de usar.

Este proyecto es importante porque representa un nuevo desafío tanto para la inteligencia artificial como para los amantes de juegos de estrategia como el ajedrez y el *Go*. Esta tesis está dirigida a los programadores de ajedrez que podrán construir sobre el programa que se desarrolló aquí. Esperamos que este trabajo sirva como base para programas futuros. También beneficia a los entusiastas del ajedrez que pueden jugar una variante interesante contra la computadora.

Además puede servir como una herramienta educativa para pedagogos del ajedrez. En la elaboración de la tesis se utilizaron las siguientes áreas de las Ciencias de la Computación:

- **Algoritmos y Estructuras de Datos:** para la elaboración del árbol de juego, los algoritmos *Minimax* y *Alfa-beta* así como a las estructuras de datos que se usaron en la lógica del juego.
- **Inteligencia Artificial:** al crear un oponente que puede tomar decisiones estratégicas.
- **Desarrollo de *Software*:** se aplicaron los principios de diseño y arquitectura de *software* para construir un programa eficiente.
- **Interacción Humano-Computadora:** para diseñar una interfaz de usuario que sea intuitiva y atractiva.
- **Gráficos por Computadora:** se usó en la visualización del tablero y las piezas de ajedrez.

En la elaboración del programa se aplicó la metodología de desarrollo *Crystal Clear*.

5.1. Heurísticas y algoritmos

El programa que se realizó puede usar dos métodos distintos para explorar el árbol de juego: *Minimax* simple y *Minimax* con poda *Alfa-beta*.

Ambos métodos se encontraron adecuados contra un oponente humano de nivel principiante y también contra los programas ya existentes. Contrario a lo esperado la versión que usa poda *Alfa-beta* no fue significativamente más rápida a la que usa *Minimax* simple, esto posiblemente se deba a que la función de evaluación no es muy compleja.

5.2. Programa

Se hicieron dos programas. La primera versión se hizo usando *Python*, usando la biblioteca *Pygame*. La segunda versión se hizo con C++ y la biblioteca *SDL*.

Al comparar ambas versiones encontramos que la versión hecha en C++ es más rápida, además la biblioteca *SDL* funciona mejor, no hay tiempo de espera perceptible al dibujar el tablero, a diferencia de lo que pasa en *pygame*.

5.3. Trabajo a futuro

- **Separar la interfaz de usuario del motor:** en el programa actual la interfaz de usuario y el motor del juego están implementados por el mismo programa. Esto es contrario a la práctica estándar, que es tener un motor de juego sin interfaz de usuario. La interfaz de usuario no se usa en torneos, solamente se usa de forma recreativa y la maneja un programa distinto. Los programas de interfaz de usuario normalmente los hace un desarrollador diferente que hace un programa de interfaz de usuario compatible con una multitud de motores con los que se comunica por un protocolo estándar.

- **Protocolo de comunicación:** los motores de ajedrez estándar tienen un protocolo estándar para comunicarse entre ellos, esto facilita las partidas entre distintos motores. Es necesario crear un protocolo similar para el ajedrez hexagonal.

- **Mejorar la función de ponderación:** en este momento la función de evaluación tiene solamente dos características: el valor de las piezas y la seguridad del *rey*. Se puede mejorar incluyendo otras características como estructura de los *peones*, control del centro de tablero, etc. Otra posibilidad es sustituir la función artesanal y en vez de esto usar una función entrenada con una red neuronal.

- **Libro de aperturas:** las fases iniciales del juego son las más difíciles para una IA, esto es debido a que el gran número de piezas sobre el tablero hace que el factor de ramaje en la primera fase del juego sea muy grande. Esta desventaja se aminora usando un libro de aperturas, este consiste en un compendio de las aperturas más comunes del ajedrez. Estas aperturas ya tienen sus posiciones analizadas así que el motor ya no tiene que analizarlas otra vez.

  El desafío aquí consiste en determinar cuáles aperturas irán en el libro y hasta qué profundidad, este problema es menor en el ajedrez tradicional pues sus aperturas han sido estudiadas extensivamente.

- **Estudio y comparación del factor de ramaje del árbol de juego:** hace falta estudiar más los árboles de juego generados por el ajedrez hexagonal, y compararlos con los árboles de juego generados por el ajedrez. Es especialmente importante encontrar el factor de ramaje promedio. Intuitivamente sabemos que el factor de ramaje es mayor que el del ajedrez tradicional. ¿Pero qué tanto? Esto puede determinar

si el algoritmo *Minimax* con poda *Alfa-beta* es una buena solución para el ajedrez hexagonal o de lo contrario sería mejor usar un algoritmo diferente como el algoritmo *MCTS*.

Existen dos métodos para lograr hacer este estudio, el primero sería analizar una base de datos de partidas existentes, serán especialmente interesantes las partidas de alto nivel. Sin embargo no tengo conocimiento de que exista tal base de datos de forma computarizada.

La otra opción es hacer que el motor juegue miles de partidas contra sí mismo y luego calcular el factor de ramaje promedio. Sin embargo este último método puede que no genere partidas representativas.

- **Motor con redes neuronales:** una nueva generación de motores para juegos de mesa son los que combinan las redes neuronales con el auto-juego y el algoritmo de Árbol de Búsqueda *Monte Carlo*. Sería interesante construir un motor alternativo que use éste método.

ANEXO A

¿CÓMO EJECUTAR EL PROGRAMA?

Esta es la estructura del programa.

```
ajedrez_hexagonal
├── c_plus
│   └── hex1.exe
├── hex1
├── IMAGENES
├── IMAGENES.py
├── ver_py
│   └── juego.py
├── launcher.py
└── requirements.txt
```

El primer paso es instalar los paquetes requeridos. Para eso se usa la instrucción:

```
pip install -r requiremnts.txt
```

Para iniciar el programa se debe ejecutar con *python* el archivo `launcher.py`. Se necesita respetar la estructura para que `launcher.py` pueda encontrar los archivos necesarios.

A continuación se explicará la funcionalidad de cada carpeta.

- `c_plus`: Contiene la versión C++ del programa,(`hex1.exe`). Así como las bibliotecas necesarias.
- `hex1`: Contiene el código fuente de la versión C++.
- `IMAGENES`: Contiene las imágenes que necesita `hex1.exe` para funcionar.
- `IMAGENES_py`: Contiene las imágenes que necesita la versión *python* para funcionar.
- `ver_py`: Contiene el código fuente de *python*.

- `launcher.py`: Este es un pequeño programa que permite elegir las opciones del juego
- `requirements.txt`: Contiene los paquetes de python requeridos.

BIBLIOGRAFÍA

[Abr89] Bruce Abramson. «Control Strategies for Two-Player Games». En: *ACM Computing Surveys* 21.2 (jun. de 1989). Number: 2.

[Bec+] Kent Beck et al. *Manifesto for Agile Software Development*. URL: https://agilemanifesto.org/ (visitado 17-11-2023).

[CK17] Jean-Louis Cazaux y Rick Knowlton. *A world of chess*. Jefferson, NC: Mcfarlane & Company, Inc., 2017.

[Coc04] Alistair Cockburn. *Crystal Clear A Human-Powered Methodology for Small Teams*. Addison Wesley Professional, 2004. 336 págs.

[Den08] Gary M. Denelishen. *The Final Theory of Chess*. Berea, Ohio: Phillidor Press, 2008.

[Dog20] Peter Doggers. *Leela Chess Zero Beats Stockfish 106-94 in 13th Chess.com Computer Chess Championship*. Chess.com. 18 de abr. de 2020. URL: https://www.chess.com/news/view/13th-computer-chess-championship-leela-chess-zero-stockfish.

[Eal85] Richard Eales. *CHESS The History of a Game*. Londres: B T Batsford, 1985.

[Ens11] Nathan Ensmenger. «Is chess the drosophila of artificial intelligence? A social history of an algorithm». En: *Social Studies of Science* 42.1 (2011). Number: 1, págs. 5-30.

[Ewa12] Timo Ewalds. «Playing and Solving Havannah». Master of Science. Edmonton, Alberta: University of Alberta, 2012.

[GBC16] Ian Goodfellow, Yoshua Bengio y Aaron C. Courville. *Deep Learning*. Adaptative Computation and Machine Learning. Cambridge, MA: MIT Press, 2016.

[Gui] *Oral History of Guido van Rossum*. Col. de Guido van Rossum. 26 de jul. de 2018. URL: https://www.youtube.com/watch?v=Pzkdci2HDpU (visitado 03-09-2023).

[Her18] Jaap van den Herik. «Computer chess: From idea to DeepMind». En: *ICGA journal* 40 (2018), págs. 160-176.

[HH21] Guy Haworth y Nelson Hernandez. «The 20th Top Chess Engine Championship: TCEC20». En: *ICGA journal* 43.1 (mayo de 2021). Number: 1, págs. 62-73.

[HS15] Sebastian Hungerecker y Malte Schmitz. «Concepts of Programming Languages – CoPL'15». En: *Bachelor Seminar Informatics*. Bachelor Seminar Informatics. Lübeck, Alemania, 2015. URL: https://www.isp.uni-luebeck.de/sites/default/files/lectures/ws_2015_2016/CS%203702%20BachSemInf%2C%20%2C%20CS%203703%20BachSemMI%2C%20%2C%20CS%205480%20SemSSE%2C%20%2C%20CS%205840%20SemiEngl/copl-proceedings.pdf.

[HW92] David Hooper y Kenneth Whyld. *The Oxford Companion to Chess*. Gran Bretaña: Oxford University Press, 1992.

[Idr13] Idris. *Pygame for Python Game Development How-to*. Birmingham, UK.: Packt Publishing, 2013.

[KM75] D. E. Knuth y R. W. Moore. «An Analysis of Alpha-Beta Pruning». En: *Artificial Intelligence* 6 (1975).

[Kum06] Kumar. «Enforcing the Gnu Gpl». En: *University of Illionois Journal ofLaw, Technology and Policy* 1 (oct. de 2006).

[Kur20] Andrey Kurenkov. «A Brief History of Neural Nets and Deep Learning». En: *Skynet Today* (2020). URL: https://skynettoday.com/overviews/neural-net-history.

[Lin92] Long-Ji Lin. «Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching». En: *Machine Learning* 8 (1992), págs. 293-321.

[Mar90] T. A. Marsland. «Computer Chess Methods». En: *Encyclopedia of Artificial Intelligence*. Dic. de 1990.

[Mit13] S. Mitchell. *SDL Game Development*. Birmingham, UK.: Packt Publishing, 2013.

[Mon13] Devin Monnens. «I commenced an examination of a game called 'tit-tat-to'": Charles Babbage and the "First" Computer Game». En: *Proceedings of DiGRA 2013*. 2013 DIGRA International Conference: Defragging game studies. Atlanta, Estados Unidos, 2013.

[MPT22] Shiva Maharaj, Nick Polson y Alex Turk. «Chess AI: Competing Paradigms for Machine Intelligence». En: *Entropy* 24.4 (abr. de 2022). Number: 4 Publisher: MDPI AG, pág. 550. DOI: 10.3390/e24040550. URL: https://doi.org/10.3390%2Fe24040550.

[Mur12] H. J. R Murray. *A History of Chess*. La edición original de 1913. New York: Skyhouse Publishing, 2012.

[Neu28] John von Neuman. «Zur Theorie der Gesellschaftspiele». En: *Mathematische Annalen* 100 (1928), págs. 295-320.

[New03] Monty Newborn. *DeepBlue: An Artificial Intelligence Milestone*. New York: Springer Verlag, 2003.

[New11] Monty Newborn. *Beyond Deep Blue. Chess in the Stratosphere*. Londres: Springer Verlag, 2011.

[New75] Monroe Newborn. *Computer Chess*. ACM Monograph Series. Academic Press, 1975.

[Nil] N. J. Nilsson. *The Quest for Artificial Intelligence. A History of Ideas and Achievements*. Web Version. Cambridge University Press.

[Pea82] Judea Pearl. «The Soluction for the Branching Factor of the Alpha-Beta Pruning Algorithm and Its Optimality». En: *Communications of the ACM* 25.8 (ago. de 1982), págs. 559-564.

[Pea84] J. Pearl. *Heuristics intelligent search strategies for computer problem solving*. London: Addison-Wesley, 1984.

[Pla+14] Aske Plaat et al. «SSS* = Alpha-Beta + TT». En: *CoRR* abs/1404.1517 (2014). arXiv: 1404.1517. URL: http://arxiv.org/abs/1404.1517.

[Pre00] Lutz Prechelt. *An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program*. 2000.

[Pri07] D. B. Pritchard. *The Classified Encyclopedia of Chess Variants*. Harpenden, England: John Beasley, 2007.

[Pur+16] Munish Puri et al. «Introduction to Artificial Neural Network(ANN) as a Predictive Tool for Drug Design, Discovery, Delivery, and Disposition: Basic Concepts and Modeling». En: *Artificial Neural Network for Drug Design, Delivery and Disposition*. Estados Unidos: Academic Press, 2016.

[Qui10] Frank Quisinsky. *Interview with Tord Romstad (Norway), Joona Kiiski (Finland) and Marco Costalba (Italy)*. Schachwelt. 28 de mar. de 2010. URL: https://www.schach-welt.de/schach/computerschach/interviews/romstad-kiiski-costalba-eng (visitado 02-09-2023).

[RN95] Stuart J. Russell y Peter Norvig. *Artificial Intelligence A Modern Approach*. Englewood, Nueva Jersey: Prentice Hall, 1995.

[Sam] A. L. Samuel. «Some Studies in Machine Learning Using the Game of Checkers. II Recen Progress». En: ().

[SB68] James R. Slagle y Philip Bursky. «Experiments With a Multipurpose, Theorem-Proving Heuristic Program». En: *Journal of the Association for Computing Machinery* 15.1 (ene. de 1968), págs. 85-99.

[SB98] Richard S. Sutton y Andrew G. Barto. *Reinforcement Learing: An Introduction*. MIT Press. Cambridge MA, 1998.

[Seb96] Robert W Sebesta. *Concepts of programming languages*. 3. edition –. Reading, MA: Addison-Wesley, 1996. 634 págs.

[Sha89] J. Shaeffer. «The History Heuristic and Alpha-Beta Search Enhancements in Practice». En: *EEE Transactions on Pattern Analysis and Machine Intelligence on Pattern Analysis and Machine Intelligence* 11.11 (nov. de 1989). Number: 11.

[She11] David Shenk. *The Immortal Game, A History of Chess*. Souvenir Press. 2011.

[Sil09] David Silver. «Reinforcement Learning and Simluation-Based Search in Computer Go». Doctor of Philosophy. Edmonton, Alberta: University of Alberta, 2009.

[Sil+17] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. DOI: 10.48550/ARXIV.1712.01815. URL: https://arxiv.org/abs/1712.01815.

[Sil19] Albert Silver. *Standing on the shoulders of giants*. chessbase. 2019. URL: https://en.chessbase.com/post/standing-on-the-shoulders-of-giants.

[Som18] James Somers. «How the Artificial-Intelligence Program AlphaZero Mastered Its Games». En: *The New Yorker* (28 de dic. de 2018). URL: https://www.newyorker.com/science/elements/how-the-artificial-intelligence-program-alphazero-mastered-its-games.

[Sre18] S. R. Sreerag. «Evaluation of Strategy by AlphaZero and StockFish on Chess Game». En: *Journal of Basic and Applied Engineering Research* 5.5 (2018). Number: 5; July-September, págs. 443-444.

[Str] Bjarne Stroustrup. *A history of C++: 1979-1991*. 1995.

[Str86] B. Stroustrup. «An Overview of C++». En: *SIGPLAN Notices* 21.10 (oct. de 1986). Number: 10.

[TG07] Veselin Topalov y Zhivko Ginchev. *Topalov-Kramnik. 2006 World Chess Championship. On the Edge of Elista*. Milford, Connecticut: Rusell Enterprises, 2007.

[Woo21] Michael Wooldridge. *A Brief History of Artificial Intelligence*. Nueva York: Flatiron Books, 2021.

[WR17] Haohan Wang y Bhiksha Raj. *On the Origin of Deep Learning*. _eprint: 1702.07800. 2017.

[YT18] Georgios N. Yannakakis y Julian Togelius. *Artificial Intelligence and Games*. Springer, 2018.

[Yuz06] Erik Yuzwa. *Game Programming in C++: Start to Finish*. Boston, MA: Charles River Media, 2006.