Universidad de Sonora

División de Ciencias Exactas y Naturales

Departamento de Matemáticas

Desarrollo de un motor de ajedrez. Algoritmos y heurísticas para la reducción del espacio de búsqueda.

Tesis que para obtener el título de

Licenciado en Ciencias de la Computación

presenta

Omar Edgardo Lugo Sánchez

Director de tesis: Julio Waissman Vilanova

Hermosillo, Sonora, mayo de 2010.

Dedicatoria

A Dios, mi familia y Amigos

Agradecimientos

Si bien, esta tesis de licenciatura requirió mucho trabajo, esfuerzo y dedicación por parte del autor y su director de tesis, no hubiese sido posible sin el apoyo de las personas que a continuacion mencionaré:

Antes que nada, agradezco a Dios, por iluminar mi mente, mi vida y sobre todo por haber puesto en mi vida a todas aquellas personas que han sido un soporte y compañia durante mis estudios.

A mi familia por procurar mi bienestar siempre. A mi madre y padre por estar siempre conmigo, ya que a ellos les debo cada logro. Han sido un apoyo incondicional durante toda mi vida, guiando y aconsejándome en cada paso que doy, por ello les estoy profundamente agradecido. A mis hermanos y amigos ya que el ánimo, apoyo y alegría que me brindan me dan la fortaleza necesaria para seguir adelante.

Un agradecimiento muy especial a mi director de tesis Dr. Julio Waissman Villanova, ya que sin su dirección, dedicación, motivación, amistad, paciencia y apoyo esta tesis no hubiese sido terminada. Un excelente profesor y amigo, ha sido una de las personas que más a influenciado mi formación académica. De igual manera mi más sincero agradecimiento al Dr. Pedro Flores Pérez ya que trabajar con él ha ayudado en mi formación profesional.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Historia del ajedrez computacional

El periodo de tiempo comprendido entre 1949 y 1959 es considerado como el nacimiento del ajedrez computacional. En el año de 1949 el matemático Claude Shannon, escribió el artículo *Programming a Computer for Playing Chess* [37]. Este artículo fue el primero en contener los principios básicos para hacer un programa que jugara ajedrez, los cuales aún son usados en los programas de ajedrez de la actualidad. Igualmente, Shannon describió las posibles estrategias de búsqueda necesarias debido a la enorme complejidad del ajedrez (estas estrategias serán descritas es la sección 2). Un año después, en 1950, el matemático ingles Alan Turin[44] inventó un algoritmo enfocado a enseñar a una máquina a jugar ajedrez, pero en ese tiempo la computadora que pudiera ser programada con dicho algoritmo aún no existía. Turing llevó a cabo su algoritmo usando una pluma y un pedazo de papel para jugar en contra de uno de sus colegas. Si bien el programa perdió, este partido dio comienzo al ajedrez computacional.

En el mismo año, John Von Neumann creó una enorme máquina llamada MINIAC, la cual podía calcular alrededor de 100,000 instrucciones por segundo, y también era programable. Su tamaño y poder de cálculo eran inmensos para ese tiempo, ya que fue construida con el objetivo de realizar los cálculos necesarios para poder producir bombas atómicas. En lugar de inmediatamente trabajar en los cálculos para las bombas, los científicos empezaron a experimentar con la máquina. Una de las primeras cosas que hicieron fue escribir un programa de ajedrez para la computadora: fue para un tablero reducido de 6x6 y no contaba con alfiles. El programa necesitaba alrededor de 12 minutos a una profundiad de 4 para realizar un movimientos, y de haber contado con los alfiles hubiera requerido alrededor de 3 horas. El programa jugó

solamente 3 partidas: una contra él mismo (ganaron blancas); el segundo contra un jugador fuerte (el juego duró 10 horas y terminó con la victoria del maestro); finalmente el programa jugó contra una joven mujer que aprendió el juego una semana antes, el programa ganó en solamente 23 movimientos. Fue la primera vez que un humano perdía ante una computadora en un juego de habilidad intelectual[32].

Durante 1958 un grupo de científicos americanos de la Universidad Carnegie-Mellon, desarrollaron mejoras al algoritmo de búsqueda básico al cual llamaron algoritmo α–β (es descrito en el capítulo 3). Este algoritmo permitió búsquedas a mayores profundidades, lo que facultaba a las computadoras jugar ajedrez más eficientemente. Durante la década siguiente, Ken Thompson contruyó una máquina cuyo único objetivo era jugar ajedrez llamada Belle [10]. Ésta, era mucho más fuerte que cualquier programa de ajedrez existente y conservó este título entre todos los progamas que jugaban ajedrez por un gran periodo.

El progreso en el ajedrez computacional, fue enfocado en esa época principalmente en incrementar el poder de cómputo de los programas, a través de hardware dedicado. El poder de juego de los programas de ajedrez ya se podría equiparar al de varios maestros, sin embargo, el juego era aún dominado por los humanos. Al final de 1980, un grupo independiente de estudiantes crearon su propio programa de ajedrez llamado *Deep Thought*, el cual fue uno de los más fuertes durante esos años. *Deep Thought* fue el prototipo usado para crear *Deep Blue*, el cual consistía en hardware especializado para jugar ajedrez, y podía calcular 200 millones de posiciones por segundo, en comparación a las 5 jugadas por segundo en promedio que puede analizar una gran maestro, esto da una idea de lo eficientes que son los humanos para jugar ajedrez, sin contar con una capacidad de cálculo tan grande como las computadoras. *Deep Blue* fue construida por un equipo de IBM con el solo propósito de derrotar al campeón reinante en esa época del ajedrez: Garry Kasparov.

En 1996 se llevó a cabo un primer encuentro contra Garry Kasparov el campeón del mundo. El encuentro terminó con una aplastante victoria para Garry Kasparov. Un año más tarde y con una nueva versión mejorada de *Deep Blue*, la computadora derrotó a Garry Kasparov. Fue la primera vez que un programa de ajedrez derrotaba al campeón mundial, lo que generó gran expectativa y sorpresa en su época. Años después numerosos encuentros entre hombre y máquina fueron llevados a cabo, principalmente contra los campeones del mundo, en los que destaca el encuentro entreVladimir Kramnik vs *Deep Fritz* en el 2002, el cual terminó con una victoria para la máquina. Es importante resaltar que estos programas corrían en computadoras personales y por lo tanto no contaban con el enorme poder de cálculo de sus predecesoras como *Deep Blue*, lo que significa que la ventaja debida a un hardware especializado puede ser

remplazada por el diseño y aplicación eficiente de algoritmos de búsqueda sofisticados.

En la actualidad el programa más fuerte es el programa *Rybka*, el cual compite al mismo nivel de los grandes maestros del ajedrez y ha ganado cada uno de los torneos donde ha participado, incluso ha derrotado a grandes maestros, aún con *handicap* en su contra. Los días en que los humanos superaban a las máquinas en un juego de habilidad mental como el ajedrez han quedado atrás, y ahora éstas compiten a un nivel igual o superior a los grandes maestros en este juego. Esto es posible gracias al esfuerzo invertido en la investigación y desarrollo de algoritmos y heurísticas de búsqueda sofisticados para este tipo de problemas, durante los pasados 60 años, sobre todo por parte de la inteligencia artificial.

1.2. Conceptos básicos

El ajedrez es un juego de suma–cero, determinista, finito y de información completa. Suma-cero significa las metas de los competidores son contrarias, una victoria para uno es una derrota para el otro. Supongamos que una posición puede ser evaluada con un valor de 20 para un jugador, entonces el valor para su oponente debe ser -20. Determinista significa que para cada estado y cada acción que tomemos sabemos siempre cual será el estado siguiente. Finito, ya que los juegos no pueden extenderse por siempre, debido a las reglas del ajedrez como: las 3 repeticiones de una posición resulta en un empate y también a la regla de los 50 movimientos, que dice que si en 50 movimientos no hay una captura o movimiento de peón entonces es un empate. Información completa significa que no hay información oculta o incertidumbre como en un juego de cartas. Adicionalmente el juego es secuencial y se juega por turnos.

Un juego de dos jugadores como el ajedrez, puede ser representado por un árbol, donde cada nodo es una posición en el tablero, comenzando desde la raíz con la posición inicial del juego. Los vértices entonces serían los posibles movimientos, que llevan a los siguientes nodos (posiciones). De esta manera, cada posible estado puede ser generado y agregado al árbol. Un juego, después que ha sido jugado, puede ser visto como un camino en el árbol de juego.

Hay que destacar que el árbol del juego del ajedrez es en realidad un grafo acíclico dirigido. No es cíclico, porque las reglas del ajedrez prohíben bucles eternos y como resultado, cada secuencia de posiciones es única. Debido a que diferentes caminos pueden conducir a la misma posición, en realidad es un grafo y no un árbol. Sin embargo, dado que un grafo puede ser representado como un árbol, se le refiere a menudo como un árbol y los métodos de búsqueda se usan como tal. La complejidad en el contexto del ajedrez describe el tiempo computacional o el espacio de búsqueda que es necesitado para resolver este juego. Resolver un juego requiere

una secuencia de movimientos que lleven a una posición final (ganadora o perdedora). Para darse una idea numérica del espacio de búsqueda del juego del ajedrez, algunas cifras son presentadas a continuación.

En cada posición en un juego de ajedrez existen alrededor de 30 a 40 movimientos legales [37]. Los 20 vértices del nodo raíz son los primeros movimientos posibles para el jugador de piezas blancas. Todos los posibles estados generados por estos movimientos en el primer nivel del árbol de juego, pueden ser denominados como el espacio de búsqueda cuando se ve una jugada adelante. Para preveer una jugada en el futuro requiere que nos fijemos en las 30 configuraciones resultantes (caso promedio), 2 jugadas requiere otros 30 movimientos por cada configuración anterior, por lo que para este nivel serían $30^2 = 900$ posiciones posibles para el espacio de búsqueda, para 6 jugadas en el futuro, el espacio contendrá $30^6 = 729,000,000$ posiciones posibles.

Tomando una media de 30 posibles movimientos por posición, y una duración de 80 jugadas (40 por cada jugador) [23], hay $30^{80} = 1.47808829 \times 10^{118}$ posibles juegos de ajedrez, esto indica que hay más posiciones en el ajedrez que átomos en el universo. Este número también es conocido como el número de Shannon, en honor al padre del ajedrez computacional. Debido a este espacio de búsqueda es tan enorme, es casi imposible generar todos los posibles estados, aunque la velocidad de cómputo sea mejorada de manera inimaginable. Esto es debido a la teoría de la medida más pequeñas de tiempo y espacio observable, el tiempo Planck y el espacio Plack que son respectivamente 10^{-43} segundos y 1.5×10^{-35}. Suponiendo que una computadora pudiera ser construida con esas dimensiones, y usando esa cantidad de tiempo para generar una posición, se podría llegar a generar todos los estados posibles de esta forma en $3.16887646 \times 10^{69}$ años. Si bien el árbol de juego existe desde el punto de vista teórico, en la práctica éste no puede ser generado completamente. Inclusive un árbol con profundidad limitada a 8 jugadas es difícil de generar desde el punto de vista práctico.

1.3. Módulos del ajedrez computacional

La publicación de 1950 de Shannon *Programming a Computer for Playing Chess* aún describe las bases del ajedrez computacional actual, ya que desde entonces no mucho ha cambiado respecto a la forma en que las computadoras son programadas para jugar ajedrez. Un programa de ajedrez normalmente contiene 3 módulos que hacen posible jugar ajedrez para una computadora: generador de movimientos, función de evaluación y algoritmos de búsqueda.

El módulo de generación de movimientos es usado para generar recursivamente los movi-

mientos del árbol de juego (aristas). La función de evaluación asigna un valor a las hojas del árbol para que, mediante los métodos de búsqueda, se encuentre el camino a la mejor posición en el árbol. Lógicamente, cuando una posición ganadora es encontrada (estado terminal), la función de evaluación tiene que retornar el valor más alto posible. Los valores para el oponente deberían ser exactamente estos valores pero negados, ya que es un juego de suma-cero. Todos los demás estados no ganadores o no finales obtendrán un valor numérico entre estos valores máximos, debido a que los algoritmos de búsqueda usan esta información para encontrar el mejor movimiento.

En su publicación, Shannon distinguió entre 2 estrategias diferentes que debían seguirse en las búsquedas para el juego de ajedrez, la de tipo A, donde todas las combinaciones de movimientos son examinadas hasta una profundidad dada, y la de tipo B, en las cual sólo los movimientos prometedores son explorados con la esperanza de reducir las ramificaciones del árbol de búsqueda. Estas dos estrategias son estudiadas en los siguientes capítulos.

1.4. Objetivo y metas del trabajo de tesis

El objetivo de este trabajo fue desarrollar un motor de ajedrez, denominado *Buhochess*, el cual sirviera como una base para el estudio e implementación de algunos de los métodos de búsqueda más importantes en juegos deterministas de suma cero. Si bien, para la implementación del motor de ajedrez *Buhochess* fue necesario implementar los tres módulos que contiene todo motor de ajedrez, el trabajo se centra principalmente en el desarrollo y análisis de los métodos de búsquedas.

Entre las metas definidas para alcanzar el objetivo general se cuentan el desarrollo de un motor de ajedrez, desarrollado desde cero, en el cual se implementaran todos los métodos y algoritmos que utiliza un motor de ajedrez moderno. Otra de las metas fue el estudio, análisis e implementación de los principales algoritmos de búsqueda para juegos deterministas de suma cero. Otra meta del trabajo fue el estudio e implementación de algoritmos de búsqueda avanzados y de heurísticas que permitieran realizar una búsqueda con una profundidad de, al menos, 10, dentro de las restricciones de tiempo reglamentarias.

El desarrollo no se consideró completo hasta que el motor fue capaz de jugar una partida completa de ajedrez a un nivel «decente» contra un jugador humano con todas las reglas del mismo, asi como cualquier restricción de tiempo. El término «decente» es subjetivo, por lo que se consideró que el motor *Buhochess* no se consideraría terminado hasta que nunca perdiera al jugar contra un jugador humano de nivel intermedio. Esto se verificó al oponer a *Buhochess*

con los jugadores pertenecientes al club de ajedrez de la Unison. Igualmente, se consideró su nivel de juego como «decente», al ser capaz de vencer sistemáticamente a otros motores de ajedrez con un poder de juego medio, como el *Big Bang Chess* de *Apple Inc.*

1.5. Organización del trabajo

El capítulo 1 de este trabajo de introducen los conceptos básicos del ajedrez computacional y se definen los objetivos y metas del trabajo realizado. Debido a que la tesis se concentra principalmente en el desarrollo y prueba de los algoritmos de búsqueda, la mayor parte del trabajo se concentra en el desarrollo e implementación de éstos. Sin embargo el desarrollo de los módulos de generación de movimientos y función de evaluación son esenciales para el buen desempeño del motor de ajedrez. En el capítulo 2 se presentan estos componentes básicos y se muestra cómo fueron implementadas en *Buhochess*.

En el capítulo 3, se muestran los algoritmos de búsqueda básicos, así como extensiones del algoritmo para mejorar la toma de decisión. Estos algoritmos son muy sensibles a la ordenación de movimientos en la búsqueda. En el capítulo 4, se presentan las heurísticas utilizadas en la ordenación de movimientos, así como mejoras de los algoritmos básicos que permiten desarrollar búsquedas de tipo A con el menor número de nodos posible. Sin embargo, para poder realizar una búsqueda más eficiente (sobre todo en el medio juego) en el capítulo 5 se presentan técnicas implementadas en *Buhochess* conocidas como *poda hacia adelante*, las cuales son estrategias de búsqueda tipo B. Con el fin de analizar en forma experimental el beneficio de cada algoritmo y heurística implementada, en el capítulo 6 se presentan algunos ejemplos ilustrativos. Por último, se presentan las conclusiones y posibles trabajos futuros.

Capítulo 2

Generador de movimientos y función de evaluación

2.1. Introducción

Todo motor de ajedrez consiste de tres componentes esenciales: el primero, es el generador de movimientos, el cual es el encargado de generar los estados sucesores, este requiere una representación del tablero real de manera que la computadora pueda entenderlo; despues está el modulo búsqueda que es el que explora estos estados sucesores a través del árbol de juego; finalmente, el módulo de función de evaluación asigna un valor numérico a una posición para medir qué tan buena es. Este trabajo se centró principalmente en el módulo de búsqueda, sin embargo, los módulos de generación de movimientos y función de evaluación también fueron desarrollados para el motor *Buhochess*. En este capítulo se presentan los métodos y técnicas utilizadas para el desarrollo de ambos módulos.

2.2. Reglas del ajedrez

El ajedrez es un juego de dos jugadores, en el cual a un participante se le asigna piezas blancas y al otro las negras. Cada uno cuenta con 16 piezas al iniciar la partida: un rey, una dama, dos torres, dos caballos, dos alfiles y ocho peones. El objetivo es capturar al rey del jugador contrario. Esta captura nunca se completa, pero una vez que el rey se encuentra bajo ataque y no puede escapar, se declara *jaque mate* y el juego finaliza.

La posición de inicio del juego se muestra en la Figura 2.1. El tablero de ajedrez consistente en 64 casillas en una cuadrícula de 8x8. El jugador con las piezas blancas mueve primero, cada participante tiene un movimiento por turno y no se puede saltar el turno para mover.

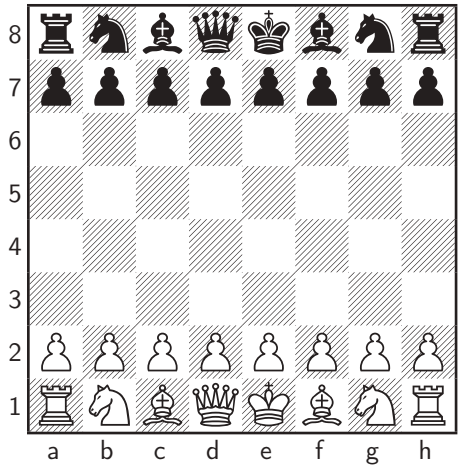

Figura 2.1: Posición inicial.

En el ajedrez existen varios tipos de movimientos: los movimientos de posición, los de captura y los movimientos especiales. Todos ellos respetan las reglas de movimiento para cada pieza [13], excepto los movimientos especiales. Los movimientos básicos de las piezas no se detallan en este trabajo por considerarlos de dominio público. Los movimientos de posición consisten en colocar una pieza en una casilla diferente de la que se encuentra y que además no esté ocupada por alguna otra. Los movimientos de captura consisten en mover una pieza propia a la casilla en la que esté la pieza del oponente, las piezas capturadas son retiradas y permanecerán fuera de juego por el resto de la partida. Los movimientos especiales requieren ciertas condiciones para llevarse a cabo:

Enroque: El rey y la torre se pueden mover en un movimiento simultáneo llamado enroque. Cuando se lleva acabo un enroque, el rey debe moverse dos casillas hacia la torre, y ésta, se coloca en la casilla adyacente al rey. Las condiciones para realizar este movimiento son las siguientes:

- El rey que hace enroque no debe haberse movido en toda la partida.
- La torre que hace enroque no debe haberse movido en toda la partida.
- El rey no debe encontrarse en jaque.

- El rey debe moverse a una casilla en la que no puede ser atacado por ninguna pieza enemiga.
- Todas las casillas comprendidas entre la torre y el rey antes de enrocar deben estar vacías.

Toma al paso: Exclusivamente de los peones, y que no es obligatoria. Ocurre cuando un peón contrario se mueve desde su casilla inicial y queda en la casilla al lado de un peón contrario en la quinta casilla. Este último puede comerlo en sentido horizontal, a derecha e izquierda, tal como lo haría diagonalmente.

Coronación: Cuando un jugador logra avanzar un peón a la octava casilla, puede pedir cualquier pieza, a excepción de otro peón o rey; y elegirá la que más le convenga.

El objetivo en el ajedrez en hacer jaque mate al rey del oponente. Cuando un rey no puede evitar estar bajo ataque enemigo se declara jaque mate y el juego finaliza inmediatamente. Sin embargo, esta no es la única forma en que puede terminar una partida de ajedrez, el juego puede terminar si se determinan que es un empate o tablas. Existen varios formas para que tablas ocurran:

- Se dice que una partida queda «tablas» cuando el rey del jugador al que le toca mover no se encuentra en jaque, pero no puede realizar ningún movimiento legal. Entonces se dice que el rey del jugador está «ahogado».
- Una partida se declara como tablas si se repite más de 3 veces una misma posición.
- Una partida se declarará como tablas si en 50 movimientos no se mueve un peón o se captura una pieza.
- Si no se cuenta con el material necesario para terminar la partida, ésta, es declarada como tablas. Por ejemplo, si se encuentra solamente rey contra rey, o rey contra rey y caballo. No existe forma de que puedan hacer jaque mate ya que el material es insuficiente.

El ajedrez normalmente está sujeto a restricciones de tiempo, éstas son determinadas dependiendo de la partida a jugar. El tiempo que le lleva a cada jugador realizar sus movimientos se contabiliza por separado. Si un jugador se pasa del tiempo dado, se pierde el juego.

2.3. Representación del tablero 0x88

La representación del tablero es una parte muy importante de cualquier programa de ajedrez, independientemente de las técnicas de búsquedas que se escojan y qué tan sofisticada sea la función de evaluación utilizada. Todos los procesos que ocurren en el juego, así como los módulos del programa se dan mediante el tablero de ajedrez, por lo que es importante escojer una representación adecuada.

La primera representación que viene a la mente es usar un simple arreglo de dos dimensiones de la forma `tablero[8][8]`. En esta representación podemos asociar la casilla *a1* a la posición `tablero[0][0]`, y así sucesivamente hasta terminar por asociar la casilla *h8* a la posición `tablero [7][7]`. Ésta, en principio parece una representación razonable; sin embargo, existen varios problemas de eficiencia asociados a esta representación. El primer problema es que para acceder a cualquier casilla del tablero del ajedrez, es necesario contar con 2 índices: uno que represente las filas y otro que represente las columnas, por lo que la tarea se complica al tener que estar manejando 2 índices para cada pieza. Además, en el momento de generar movimientos mediante esta representación se tiene que tomar en cuenta hacia donde tenemos que incrementar estos índices y las validaciones se vuelven excesivamente costosas.

En esta representación, si queremos movernos diagonalmente hacia abajo comenzando en la casilla *e4* (fila 3, columna 4), simplemente hacemos un incremento en 1 a las filas y las columnas. Si deseamos movernos diagonalmente desde la casilla *e4* hacia arriba hasta la casilla *h7*, tendríamos incrementos de la forma `tablero[3][4]`, `tablero[4][5]`, `tablero[5][6]` y `tablero[6][7]`. Es necesario ser cuidadoso en no acceder a elementos que se encuentren fuera del tablero, por lo que cada vez que hacemos un incremento o decremento a las filas o columnas, necesitamos asegurarnos que el valor de los índices estén entre 0 y 7. Esta es una desventaja, ya que el número de veces que necesitamos probar que no nos salimos de los bordes del tablero se convierten en parte significativa del ciclo generador de movimientos.

La primera idea que surge para solucionar estos problema es usar solamente un índice que represente cada pieza en el tablero, por lo que en vez de usar 2 arreglos dimensionales de 8x8 se hace uso de un solo arreglo de 64 casillas. La forma de asociar las casillas en el arreglo es de la forma *a1* a `tablero[0]`, así sucesivamente terminando en *h8* a `tablero[63]`.

Esta representación facilita la generación de movimientos, ya que podemos generar las posiciones destinos con simples incrementos al índice que representa la pieza. Por ejemplo, si queremos movernos diagonalmente desde la casilla *e4* hasta la casilla *h7*, simplemente sumamos la constante 9, y obtendríamos `tablero[28]`, `tablero[37]`, `tablero[46]` y `tablero[55]`.

Sin embargo, aunque esta representación sea más simple, rápida y facilite la generación de movimientos, aún existe el problema de tener que comprobar si en cada nuevo movimiento nos salimos de los límites del tablero. Lo ideal sería eliminar las pruebas para hacerlo más eficiente o poder hacer estas pruebas más rápidamente.

La representación 0x88 [31] pretende solucionar estos problemas al utilizar un arreglo de 128 casillas, que son simplemente 2 tableros pegados el uno al lado del otro. Uno representa el tablero real, y el otro es un tablero falso destinado a comprobar las posiciones ilegales. La representación del tablero en 0x88 se ilustra en la Figura 2.2.

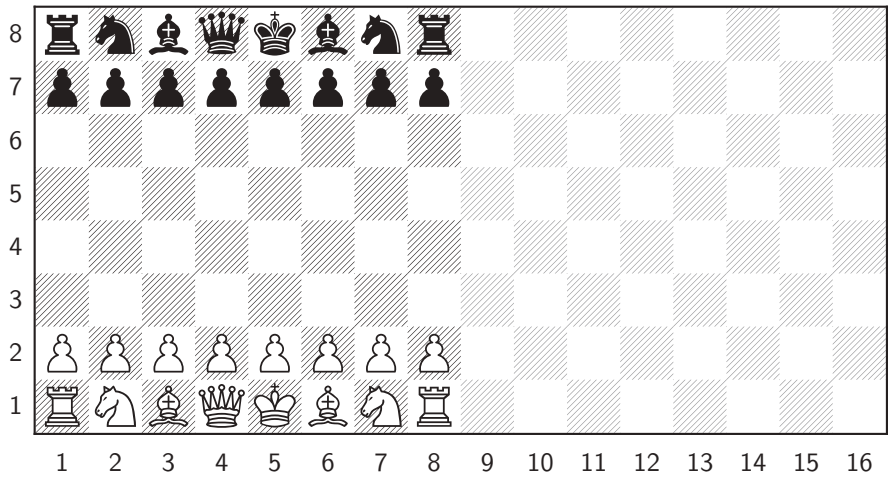

Figura 2.2: Ejemplo de la representación 0x88.

De acuerdo a la Figura 2.2, la primera fila desde la primer casilla abajo a la izquierda hasta la última casilla de la primera fila del tablero real tienen los índices de 0 hasta 7. Después vienen los índices para el tablero falso en la primera fila que van de 8 hasta 15. De igual manera, se asignan a las casillas de la segunda fila del tablero real, los índices 16 hasta 23, y para el tablero falso se asignan los índices 24 hasta 32, y así sucesivamente con todo el tablero.

La idea de representar el tablero de esta forma es que los índices que se encuentran en el tablero real tienen una característica única en forma binaria y también aquellos índices que correspondan a posición ilegales. Estas características de los índices son las que nos indicarán sí hemos salido de los bordes del tablero. Si se crea una tabla de estos índices junto a sus valores binarios se obtendría una tabla como la representada en el Cuadro 2.1.

La manera en que se utiliza esta representación es la siguente: los primeros 3 bits indican el número de fila, el cuarto indica si la posición se encuentra en el tablero falso, los 3 siguientes

bits representan el número de columna y el último bit es otro indicador para el tablero falso. El cuarto y octavo bit permite comprobar de una manera simple y rápida si una casilla se encuentra fuera del tablero, haciendo una comparacón bit a bit de la posición con el número 0x88 (88 hexadecimal). En un lenguaje tipo *C* esto podría escribirse como un simple condicional de la forma

```
if( !(indice & 0x88)  ){ //  dentro del tablero   }
```

El resultado obtenido al aplicar este operador será de 0 si la casilla se encuentra dentro del tablero real y de 1, si resultó ser una posición que se encuentra en el tablero falso, lo que resulta una posición ilegal. Además, para desplazarnos en el tablero sólo es necesario incrementar una constante al índice para obtener la nueva posición. Por ejemplo, un movimiento horizontal se logra con incrementos (o decrementos) de 1, un movimiento vertical con incrementos de 16 y un movimiento diagonal con incrementos de 15 ó 17 (diagonal a la izquierda o derecha respectivamente).

Cuadro 2.1: Representación binaria de los índices.

Tablero Real		**Tablero Falso**	
Indice	**Binario**	**Indice**	**Binario**
0	0000 0000	8	0000 1000
1	0000 0001	9	0000 1001
2	0000 0010	10	0000 1010
3	0000 0011	11	0000 1011
4	0000 0010	12	0000 1100
5	0000 0101	13	0000 1101
6	0000 0110	14	0000 1110
7	0000 0111	15	0000 1111
⋮	⋮	⋮	⋮
112	0111 0000	120	1111 1000
113	0111 0001	121	1111 1001
114	0111 0010	122	1111 1010
115	0111 0011	123	1111 1011
116	0111 0010	124	1111 1100
117	0111 0101	125	1111 1101
118	0111 0110	126	1111 1110
119	0111 0111	127	1111 1111

Otra característica de utilizar la representación 0x88 es que podemos comprobar rápidamente los ataques. Esto es importante ya que es una de las operaciones que más veces se

realiza y que consume una gran cantidad de tiempo, por lo que debe implementarse eficientemente. La comprobación de ataques se realiza por cada movimiento, ya que debe verificarse si el movimiento es legal (es decir, que no deje a nuestro rey en jaque), para cada nodo explorado durante la búsqueda. También se puede utilizar la representación 0x88 en la función de evaluación, si los ataques forman parte de un criterio de la evaluación como en el motor *BuhoChess*.

La manera en que esto funciona es debido a otra característica en los índices del tablero 0x88: si uno resta 2 índices de las casillas del tablero, uno obtendrá un valor que describe la relación entre estas dos casillas. Por ejemplo, si uno resta el índice de la casilla A, del índice de la casilla B, y se obtiene como resultado 1, se infiere que la casilla A debe encontrarse una casilla a la izquierda de B, si se obtiene 16, entonces A se encuentra una casilla encima de B en el tablero.

Para implementar esto se hace uso de un arreglo de 257 elementos (todas las posibles diferencias entre casillas), donde en cada posición del arreglo se llena con un código que describe qué piezas pueden moverse desde la posición A hacia la posición B, y por lo tanto cuáles piezas pueden realizar un ataque de A a B.

Como varias piezas pueden moverse de manera similiar por ejemplo el afil y la reina, la torre y la reina, etc., éstas pueden agruparse y destinar ciertos bits para representar estas agrupaciones. El arreglo de ataque puede llenarse con la siguiente información, por ejemplo:

- Ninguna pieza - 0: Puede darse el caso que ninguna pieza pueda moverse de esa forma para atacar a la casilla.
- Rey, reina y torre - 1: Las tres piezas pueden moverse una casilla horizontal y verticalmente.
- Reina y torre - 2 : Las piezas pueden moverse varias casillas horizontal y verticalmente.
- Rey, reina, alfil y peón blanco - 3: Todas estas piezas pueden moverse diagonalmente 1 casilla.
- Rey, reina, alfil y peón negro - 4: Todas estas piezas pueden moverse diagonalmente 1 casilla.
- Reina y alfil - 5: Pueden moverse diagonalmente varias casillas.
- Caballo - 6: El caballo es una pieza muy diferente a las demás por lo que no puede agruparse.

La manera con que se implementa esto es de la siguiente manera: se obtiene un índice restando el índice de la casilla atacante del índice de la casilla destino; enseguida se suma 128 para evitar números negativos y se consulta en el arreglo de ataques para verificar que tipo de piezas pueden moverse desde la casilla atacante a la destino. Si el tipo de pieza en el arreglo de ataques corresponde a la pieza atacante entonces se dice que esta puede atacarla. Esto se calcula utilizando la expresión

```
indice = casilla_destino - casilla_ataque + 128.
```

Consideremos la posición de la Figura 2.3. Si deseamos saber si la casilla *g1* puede ser atacada por una pieza ubicada en *c5*, se realiza la operación:

```
indice = 6 - 66 + 128 = 68.
```

Este índice se utiliza para consultar el arreglo de ataques `arreglo_ataque[68] = 5`, donde el valor de 5 representa todas aquellas piezas que pueden moverse de esa forma (alfil y reina). Como la pieza ubicada en la casilla *g1* corresponde al tipo de piezas que pueden moverse de esa forma indica que si existe un ataque.

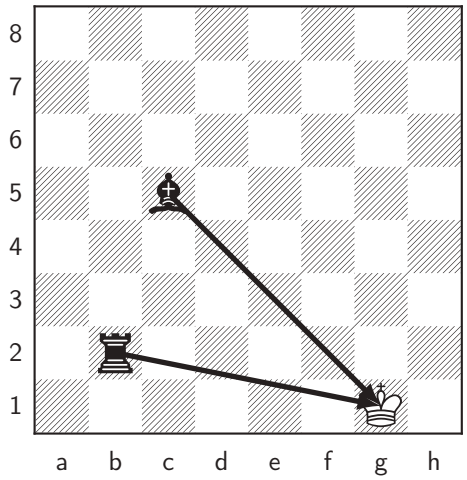

Figura 2.3: Ejemplo de ataques al rey.

Existen ocasiones en donde ninguna pieza atacará la posición deseada. Por ejemplo, si queremos verificar que la pieza en *g1* puede ser atacada por una pieza en *b2*, se obtiene el índice como `indice = 6 - 17 + 128 = 117`. Utilizando el valor obtenido se consulta el arreglo de ataques `arreglo_ataque[117]= 0`, que indica que no existe pieza que pueda moverse de esa manera para atacar dicha casilla.

Al implementar esta rutina para verificar ataques en el motor *Buhochess* el poder de cálculo del programa se incrementó cerca del 32 %.

2.3.1. Mejoras al generador de movimientos

El generador de movimientos es una de las partes fundamentales en cualquier programa de ajedrez, no solamente porque de él depende la generación de estados sucesores sino que la velocidad de cálculo de los programas (nodos por segundo explorados) está directamente relacionada a éste.

En experimentos realizados en el motor *Buhochess* se comprobó que del tiempo total empleado en la búsqueda, cerca del 60 % del tiempo es utilizado en el módulo de generación de movimientos, 30 % en la función de evaluación y el resto en otros aspectos de la búsqueda. El tiempo invertido en el generador es comprensible ya que para cada nuevo nodo explorado necesitamos generar los movimientos para esa posición.

Las primeras versiones de *Buhochess* permitía un poder de cálculo de 10 a 15 mil nodos por segundo. La capacidad de procesamiento del programa se debía a que, para guardar los nuevos movimientos de cada nodo se solicitaba memoria al sistema para guardarlos, lo que ralentizaba en gran medida el poder de cálculo. Para solucionar este problema, se utilizó una pila de tamaño suficiente para guardar todos los posibles movimientos realizados durante la búsqueda. De esta forma sólo se necesita solicita memoria al inicio de programa. Al implementar esta mejora se logró un poder de procesamiento alrededor de 140 mil nodos por segundo.

Otra mejora realizada al módulo de generación de movimientos fue que, cuando se pretendía generar los movimientos posibles para un jugador, era necesario explorar cada casilla del tablero para verificar la pieza que ocupa la casilla y así generar los movimientos para esa pieza. Para cada nodo explorado durante la búsqueda se tenía que revisar 64 veces el tablero, lo que significa una pérdida de tiempo ya que la mayoría de las casillas están vacías o tienen piezas enemigas. Además, las comprobaciones para verificar de qué tipo de pieza se trataba consumía más tiempo de cómputo. Estas operaciones no parecen consumir un tiempo valioso sin embargo, debe tenerse en cuenta que esto se realizan para cada nuevo nodo explorado y se vuelve un tiempo significativo debido a que la cantidad de nodos explorados es exponencial a la profundidad de la búsqueda.

La manera en que se evitó este problema fue mantener en memoria la posición actual de las 16 piezas para cada jugador, de forma que solamente se debe comprobar 16 veces la posición de

las piezas, a diferencia de las 64 con la representación anterior. Esta implementación aumentó el poder procesamiento cerca de 29 %, con lo que actualmente el motor *Buhochess* es capaz de explorar cerca de 210 mil nodos por segundo.

2.4. Función de evaluación

La función de evaluación es una de las partes más importantes en un programa de ajedrez, ya que determina en gran medida la fuerza de juego de cada programa. Esta heurística es usada debido a que no existe posibilidad de explorar a través de todos los movimientos posibles en el árbol de juego del ajedrez, por lo tanto es necesario hacer uso de una función de aproximación que le permita a los algoritmos de búsqueda discriminar entre diferentes posiciones. La función de evaluación es normalmente una función multi–variable que mide que tan buena es una posición de ajedrez, donde las variables de la función representan un factor que caracteriza la posición. Cabe señalar que es muy difícil indicar que características son importantes y todavía más difícil indicar qué tan importantes son cada una de las características. Entre mejor seleccionadas sean las características, mayor será el poder de juego del programa.

Una posición de ajedrez está compuesta de piezas blanca y piezas negras localizadas en ciertas posiciones en el tablero de ajedrez. Cada pieza tiene diferente importancia y se le asigna un valor a cada una dependiendo de qué tan fuerte o importante sea. La diferencia entre las piezas de un jugador y las piezas del contrario se llama *balance material*. Esta es una de las características de mayor importancia en la función de evaluación y normalmente es considerada sobre las demás, ya que todas las posibles características son en función a las piezas. Las demás características importantes que tienen que ser tomadas en consideración, como la seguridad del rey, la movilidad de las piezas, control del centro, entre otras, son llamadas estratégicas o posicionales.

Una función de evaluación está definida como $f : U \rightarrow [-Mate, Mate]$, donde U es el conjunto de todos los posibles tableros que se pueden generar en el ajedrez. Para realizar la evaluación para una posición del tablero dada n, se determinan las características que serán consideradas, y se calcula el valor de cada una. La función de evaluación más sencilla y más utilizada es:

$$f(n) = \sum_{i=1}^{N} X_i(n) \cdot V_i(n),$$

donde N es el número total de características, $V_i : U \rightarrow [-Mate, Mate]$ puede considerarse como una función de evaluación parcial tomando en cuenta una única característica de la

posición. $X_i : U \to \{0, 1\}$ es una función de decisión, que considera si el valor $V_i(n)$ será considerado en la evaluación; esto permite realizar evaluaciones diferenciadas para, por ejemplo, la media partida o finales de juego. Esta estructura de la función de evaluación permite agregar la evaluación de nuevas características conforme aumenta el conocimiento del juego, o probar varias combinaciones.

Entre mayor sea el número características que tomemos en cuenta para evaluar una posición de ajedrez más acertada será. La cantidad de características que podemos considerar puede variar desde unos cientos hasta miles, dependiendo de la robustez deseada para la función de evaluación. Sin embargo, considerar solamente las principales características ofrece una buena estimación de los estados y un buen poder de juego. Éstas, pueden ser agrupadas de la siguiente manera:

- Balance material.
- Movilidad y control del tablero.
- Seguridad del rey.
- Estructura de los peones.

En las subsecciones siguientes se describen las características consideradas en la función de evaluación del programa *Buhochess*, descritas por grupos principales.

2.4.1. Balance material

A cada pieza se le asigna un valor y estos valores son usados como heurísticas para ayudar a determinar qué tan valiosa es una pieza estratégica y tácticamente con respecto a las demás. Estos valores son importantes para determinar por cuáles y en qué momento vale la pena intercambiar piezas. Una estrategia fundamental del ajedrez indica que debemos capturar las piezas enemigas mientras se trata de preservar las propias, ya que generalmente mayor número de piezas o más poderosas significa mayores oportunidades de ganar.

Para contabilizar el valor de las piezas se utiliza un valor de medida mínimo llamado *centipawn*(cp). Los valores de las piezas, los valores posicionales y demás características se dan en términos de esta unidad. El valor de las piezas está entonces determinado por el valor del *centipawn*. En *Buhochess* se asignaron los valores de las piezas como: peón: 100 cp; caballo: 300 cp ; alfil: 310 cp; torre: 500 cp; y reina: 900 cp.

Existen varios valores para las piezas en la literatura [28], pero siempre respetando el orden mencionado. Estos valores de las piezas proveen sólo una idea básica del valor. El valor exacto de las piezas dependerá de la situación y puede variar para diferentes juegos. En algunas posiciones, una pieza bien colocada puede tener mucho más valor que el indicado por estas heurísticas.

2.4.2. Movilidad y control del tablero

El control de una casilla del tablero se traduce en más opciones de ataque y defensa para un jugador y menos para su adversario, por lo que el jugador que controle más casillas obtendrá una ventaja significativa [41], a esto se le conoce como control del tablero. Normalmente se busca controlar el centro del tablero ya que éste es el camino más rápido a cualquier parte del tablero.

Algo relacionado con el control del tablero es la movilidad de las piezas, ya que también se pretende que las piezas tengan varias opciones de ataque y de defensa. Esto se nota particularmente en las piezas que están colocadas a las orillas, ya que tienen una movilidad reducida a comparación de una pieza colocada en el centro. Las características retenidas en *Buhochess* referentes a la movilidad y control del tablero son:

Conexión de las piezas. Es necesario que las piezas se protejan entre sí, ya que al no hacerlo se permitirían puntos débiles en la posición del jugador y además podrían perderse piezas cuando se realicen intercambios. En la Figura 2.4 se muestra un tablero donde todas las piezas negras se encuentran protegidas, a excepción de un peón. Se asigna 10 cp por cada pieza protegida.

Casillas controladas por el jugador. Entre más casillas controlemos o entre mayor número de piezas enemigas ataquemos, mayores serán las opciones que podamos explotar. En la Figura 2.5 se muestran las casillas atacadas por el jugador negro. Se otorga un valor de 5 cp por cada casilla atacada.

Control del centro: Es importante tratar de controlar el centro, que son 4 casillas (*e4,d4,e5,d5*) del tablero. El centro representa una zona de suma importancia posicional ya desde él se puede llegar fácilmente a cualquier parte del tablero. Un ejemplo puede verse en la Figura 2.6. Se da un valor de 40 cp por cada peón en una de las casillas del centro, 20 cp por un caballo o alfil y 30 cp si la reina está en una de estas casillas.

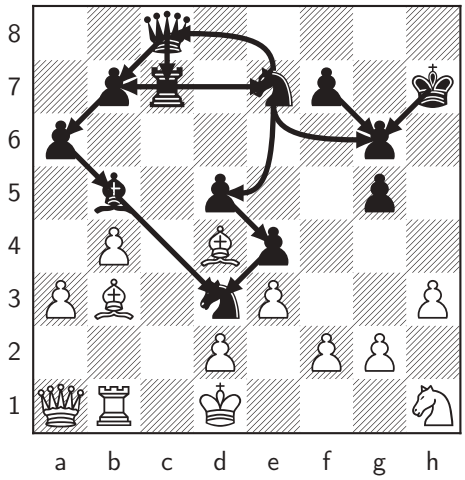

Figura 2.4: Las Piezas negras se defienden las unas a las otras.

Par de alfiles: Se considera más fuerte contar con el par de alfiles que contar con un caballo y alfil. Se asigna 25 cp si en la posición actual se cuenta con ambos alfiles.

Alfil en la primera fila: Se busca que el alfil controle más casillas en el juego pero si se conserva en la primera fila, su potencial se ve disminuido. Se da un valor de -15 cp por cada alfil en primera fila de la posición evaluada.

Torre en columna abierta: Se considera que una torre está en una columna abierta cuando no existe peones propios y enemigos en la misma columna, como se muestra en la Figura 2.7. Las torres en estas posiciónes son buenas para controlar grandes cantidades de casillas y permiten un rápido acceso al territorio enemigo. Se asigna un valor de 15 cp por cada torre en columna abierta en la posición evaluada.

Torre en columna semi-abierta: Una torre está en una columna semi-abierta cuando sólo existe un peón enemigo en la misa columna. Se busca ejercer presión en la posición enemiga atacando los peones contrarios; además, en algún momento se puede abrir la columna permitiendo ataques en el lado enemigo, tal como se aprecia en la Figura 2.8. Se otorga valor de 17 cp por cada torre en columna semi-abierta en la posición evaluada.

Conexión de torres verticalmente: Dos torres están conectadas verticalmente cuando se encuentran en la misma columna y no existe ningún tipo de pieza entre ellas. Esta configuración permite incrementar el poder de ataque de las torres, además que permite que se protejan entre sí. Se asigna un valor de 40 cp si existe una conexión de torres verticalmente en la posición evaluada.

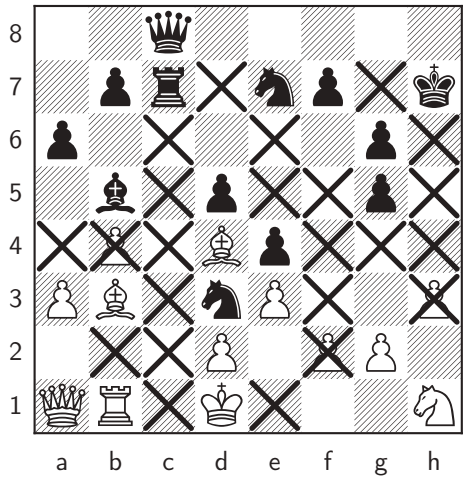

Figura 2.5: Casillas atacadas por el jugador negro.

Conexión de torres horizontalmente: Se consideran que 2 torres están conectadas horizontalmente cuando se encuentran en la misma fila y no existe ningún tipo de pieza entre ellas como se ilustra en la Figura 2.8. Se asigna un valor de 40 cp si existe una conexión de torres horizontalmente en la posición evaluada.

Torre en la séptima fila: Se coloca la torre detrás de las líneas enemigas lo cual resulta en una posición ventajosa, ya que permite atacar los peones que no pueden defenderse entre sí, además que ejerce presión sobre el rey enemigo. Se asigna un valor de 20 cp por torre en la séptima fila.

Posición de las piezas. Dependiendo de cada pieza, existen posiciones que son más ventajosas y otras que son claramente desfavorables en el tablero. Por cada pieza, se otorga un valor (positivo o negativo) dependiendo en que posición se encuentre en el tablero. En el apéndice A se muestran las matrices de valores (desde el punto de vista del jugador blanco) que se asignan a cada una de las piezas.

2.4.3. Seguridad del rey

La seguridad del rey es un criterio de suma importancia y a veces de mayor relevancia que el valor material, ya que todo el juego gira entorno en conseguir la captura del rey enemigo, y si no se le presta atención se podría perder la partida rápidamente.

Una buena evaluación de la seguridad del rey es una de las tareas más desafiantes al escribir

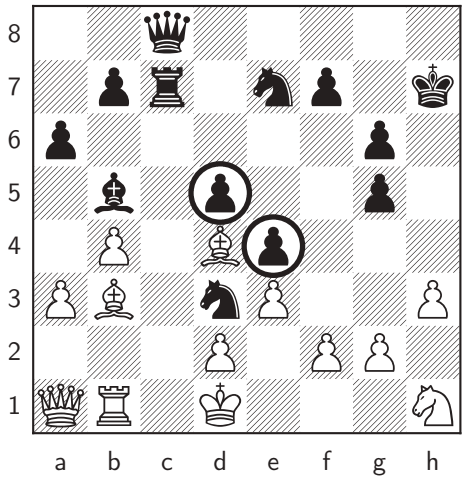

Figura 2.6: Control del centro

una función de evaluación, pero también una de las más gratificantes. Existen numerosos criterios para tratar de asegurar la seguridad del mismo y que pueden cambiar con el tiempo de juego. Normalmente éste debe encontrarse rodeado de una buena estructura de peones que funcionan a manera de escudo ante los ataques enemigos, también las demás piezas deben encontrarse cerca para tratar de defender a su rey de posibles ataques.

La seguridad del rey es especialmente importante al inicio de la partida y un poco más en la partida medio, sin embargo, una vez que la mayoría de las pieza han sido capturadas, el rey se vuelve una de las piezas más efectivas para atacar y proteger, por lo que tiene que salir a tratar de ganar terreno y defender las piezas restantes, especialmente los peones. En la función de evaluación de *Buhochess* se considerarón las siguientes características de seguridad del rey:

Enroque: Este movimiento especial permite al rey situarlo en una posición de mayor seguridad cerca de una esquina, donde la cantidad de flancos por donde puede ser atacado se ve disminuida. Si el rey puede enrocar de su lado se asigna un valor de 50 cp, del lado de la reina de 40 cp y si perdemos la oportunidad de enrocar, se asigna una penalización de -60 cp.

Amenazas: El rey no debe encontrarse en la línea de ataque de las piezas enemigas. Aunque este ataque no sea directo ya que podría interponerse una pieza, esta línea puede abrirse en un momento, dejando al rey bajo ataque u obligándolo a ciertas posiciones o movimientos desfavorables. Por cada línea de ataque al rey se asigna un valor de -8 cp.

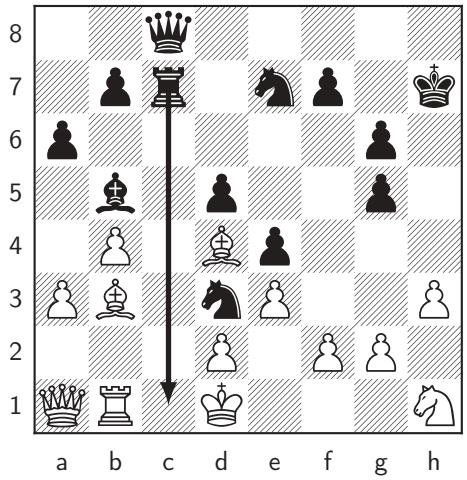

Figura 2.7: Torre en columna abierta.

Estructura de los peones del rey: Son la principal línea defensiva del rey, ya que después de enrocar sirven a manera de escudo. Si se enroca del lado del rey la evaluación se realiza con los peones cuya posición inicial es *f2,g2,h2*; si se enroca del lado de la reina, la evaluación se realizará con los peones con posición inicial *a2,b2,c2*. Aunque cada peón es importante para la garantizar la seguridad del rey, lo que se encuentras en las casillas *f2* y *c2* son un poco más flexibles al respecto y todas las penalización a estos peones se reducen a la mitad. La evaluación de esta característica se aplica a cada peón, de la siguiente manera:

- Si el peón no se movió no tiene penalización.
- Si se movió una casilla, una penalización de -10 cp.
- Si se movió más de 1 casilla, penalización de -20 cp.
- Si no existe peón en la casilla, penalización de -35 cp.
- Si no existe un peón enemigo en la misma columna que el peón, se permite una línea de ataque al rey. Se da una penalización de -15 cp.
- Si existe un peón enemigo a 1 casilla de la segunda fila, penalización de -25 cp.
- Si existe un peón enemigo a 2 casillas de la segunda fila penalización de -15 cp.
- Si no hubo enroque, se establece una penalización de -22 cp por cada línea abierta cerca de nuestro rey.

En el final de la partida cuando el material es escaso para ambos (en *BuhoChess* esto significa tener menos de 1300 cp), el rey debe salir a defender las demás piezas, por lo

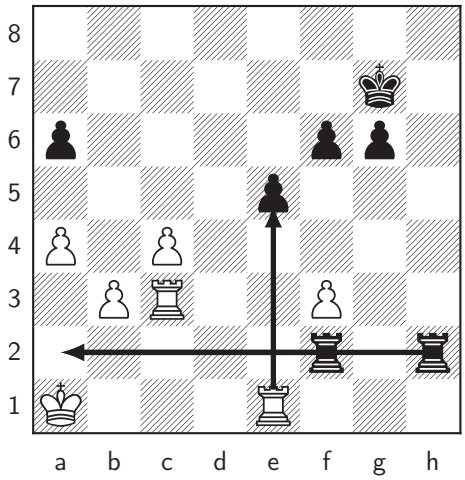

Figura 2.8: Ejemplo de las las características para las torres.

se ignora la características anterior y se reemplaza por una evaluación sobre la posición del rey, la cual se encuentra en forma de matriz en el apéndice A.

2.4.4. Estructura de los peones

Unas de las características más importantes que distinguen una posición débil de una fuerte es la estructura de los peones. Ésta, es considerada como el alma del juego por muchos estudiosos del ajedrez, y es que una buena estructura de peones pueden significar una ventaja significativa para el jugador.

La evaluación de estas estructuras juegan un parte muy importante de la función de evaluación, ya que, a diferencia de las otras piezas que pueden moverse en múltiples direcciones durante el juego, los peones están limitados a un máximo de 5 ó 6 movimientos. Los peones avanzan lenta y deliberadamente, por consecuencia la estructura de peones evoluciona lentamente y cualquier aspecto de esa estructura puede durar varios movimientos, y a veces permanecer hasta el final de la partida. La evaluación de la estructura de los peones implementada en *Buhochess* toma en cuenta las características siguientes:

Peones pasados. Se conoce así al peón que no tiene peones enemigos que prevengan que éste avance hasta la octava fila. Por ejemplo, si no existen peones opuestos que se encuentren en la misma fila que el peón o en las filas adyacentes a éste que puedan atacarlo. Un peón pasado sería el que se encuentra en la casilla *f7* de la Figura 2.9. Un peón pasado

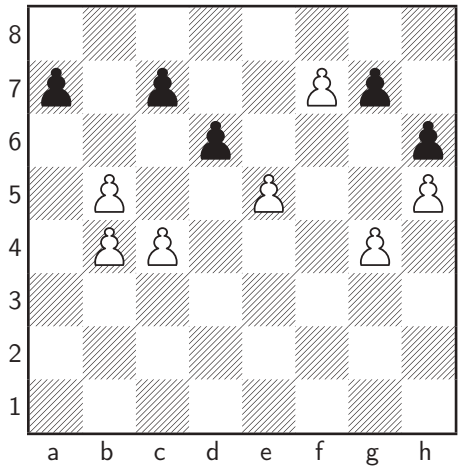

Figura 2.9: Ejemplo de las características de las estructuras de peones.

puede ser de gran ventaja ya que solamente las piezas contrarias pueden prevenir que estas corone, reduciendo su potencial, ya que estarán enfocadas en evitar su avance. Se asigna un valor de 20 cp por cada peón pasado en la posición evaluada.

Peones aislados. Estos peones son débiles por 2 razones: la primera es que las piezas que los atacan usualmente tienen mayor flexibilidad que aquellas que las defienden. La segunda razón es debido a que no tienen peones adyacentes que los defiendan, por lo que pueden ser atacados por varias piezas enemigas y resultan piezas fáciles de capturar. El peón en la casilla *e5* es un ejemplo de un peón asilado mostrado en la Figura 2.9. Se asigna un valor de -20 cp por cada peón aislado en la posición evaluada.

Peones retrasados. Estos peones son los que se encuentran adyacentemente atrás de otro peón del mismo color, éstos son una desventaja posicional, ya que son difíciles de defender al encontrarse hasta atrás de los demás peones. Esto se observa en el peón en la casilla *g4* en la Figura 2.9. También son una desventaja debido a que el oponente puede colocar una pieza enfrente del peón (usualmente un caballo), sin ningún riesgo de que un peón lo ataque. Se evalúa con -8 cp por cada peón retrasado en la posición evaluada.

Peón doblado. Se considera que un peón está doblado si dos peones del mismo color se encuentran en la misma columna, como el que se encuentra en la casilla *b5* en la Figura 2.9. Los peones se doblan solamente cuando capturan en la columna donde se encuentra una pieza enemiga. En la mayoría de los casos los peones doblados son considerados una debilidad debido a la incapacidad de éstos para defenderse entre sí. Se asigna un valor

de -10 cp por cada peón doblado en la posición evaluada.

2.5. Comentarios finales

En este capítulo se presentaron dos de los módulos principales de un motor de ajedrez: el módulo de generación de movimientos y la función de evaluación. La implementación del módulo generador de movimientos está estrechamente relacionado con las estructuras de representación de tablero, y debido a que este módulo es uno de los que más consume tiempo, es necesario implementarlo con estructuras eficientes. Se mostraron los métodos retenidos y las consideraciones que se hicieron en el desarrollo práctico del módulo generador de movimientos del motor *Buhochess*.

La función de evaluación determina la forma y el poder de juego de los programas. Elegir qué características retener para la evaluación no es una tarea trivial y requiere mucho conocimiento sobre el tema. Otra tarea compleja es asignar los valores de cada característica, así como la estructura de la función de evaluación. Para el motor *Buhochess* se decidió por emplear una función lineal y se establecieron tanto las características como el valor de éstas a través de una búsqueda bibliográfica, experiencia del autor, así como la experimentación del motor al competir con diversos jugadores. Se presentan las características retenidas e implementadas, así como los valores asignados a éstas.

Capítulo 3

Algoritmos de búsqueda

3.1. Introducción

Para realizar una búsqueda en el juego del ajedrez, éste, puede ser representado mediante un árbol, en el cual los nodos representan posiciones del tablero y las ramas simbolizan los movimientos posibles en cada posición. El objetivo de la búsqueda es encontrar un camino desde la raíz del árbol hasta un nodo terminal que tenga el valor de utilidad más alto posible. En general se espera ganar el juego, lo que se especifica por un valor numérico al que llamaremos $Mate$. De no ser posible, se espera al menos obtener un valor de utilidad de $Tablas$ o empate. Esta es la idea básica de todos los algoritmos de búsqueda en el juego del ajedrez. En este capítulo se dará una introducción al algoritmo de búsqueda básicos *Minimax*, y una mejora a él llamado poda α–β. También se presenta la búsqueda *Quiscence* y extensiones de búsqueda.

3.2. Algoritmo de búsqueda *Minimax*

Para realizar esta búsqueda se consideran 2 jugadores, a los que se les denomina *Max* y *Min* [42], los cuales son contrincantes y juegan de manera alternada. Igualmente, se asume que ambos jugadores seleccionarán sus movimientos *siempre de la mejor manera posible.*

La idea de la búsqueda consiste en asignar la posición actual del tablero como nodo raíz, el generador de movimientos para encontrar los nodos del siguiente nivel de profundidad, y así consecutivamente. Sin embargo, como no es posible en términos prácticos generar todos los nodos del árbol de juego, se generan los nodos hasta un nivel de profundidad d especificado

de antemano. A continuación se aplica la función de evaluación a todas las posiciones finales obtenidas y se propagan estos valores a los niveles superiores del árbol con el fin de elegir únicamente el primer movimiento que lleve hasta la posición cuya evaluación entregue el valor mayor.

Así el valor de la función de evaluación no deberá ser nunca mayor a $Mate$, debido a que este valor implica que, se ha ganado el juego, o solamente existen movimientos que lleven a una victoria. De la misma forma, el valor de la función de evaluación no podrá ser menor a $-Mate$ ya que el ajedrez es un juego de *suma cero*, por lo que la meta de los jugadores es contraria.

Durante la búsqueda consideramos la alternancia en la participación del juego por parte de los jugadores. Para cada nivel del árbol que corresponda al jugador *Max*, éste tratará de maximizar sus ganancias en ese nodo y escogerá el movimiento entre sus posibles acciones que lo lleven a obtener un valor máximo de utilidad. En cada nivel que corresponda el jugador *Min* éste también tratará de maximizar su ganancia, o lo que es lo mismo, minimizar las ganancias de su contrincante en ese nodo. Así, el jugador *Min* realizará los movimientos que lleven siempre el mínimo valor posible de utilidad. Es de notar que la utilidad, obtenida a través de la función de evaluación, siempre se considerará en relación al jugador *Max*.

Estas operaciones se hacen de forma recursiva hasta explorar todo el árbol a una profundidad d. Así, el jugador *Max* siempre escogerá el movimiento que lo lleve a obtener un valor máximo de utilidad entre todos los valores que permitan los nodos *Min* sucesores, mientras que en los nodos del jugador *Min*, se escogerá el movimiento que permita obtener el mínimo valor de utilidad posible entre todos los nodos *Max* sucesores de éste. El resultado será un camino desde la raíz del árbol hasta un nodo a profundidad d que permita obtener la mayor utilidad posible para el nodo *Max* raíz. A este camino se le conoce como *variación principal*.

El algoritmo *Minimax* garantiza encontrar siempre el movimiento óptimo de acuerdo a una función de evaluación establecida, una profundidad d fija y asumiendo que, a su vez, el jugador *Min siempre juega de manera óptima* en el mismo sentido que el jugador *Max*. Sin embargo, la complejidad del algoritmo *Minimax* está directamente relacionada con el número de nodos a expandir, la cual está dada por $\mathcal{O}(\omega^d)$, donde ω es el máximo número de jugadas posibles en una posición y d es la profundidad de la búsqueda.

El árbol de la Figura 3.1 ejemplifica el algoritmo *Minimax*. En cada nivel que corresponda el jugador Min, éste seleccionará los movimientos que lo lleven a minimizar lo más posible las ganancias de su adversario. Por ejemplo, los nodos a y b son posibles movimientos para Min;

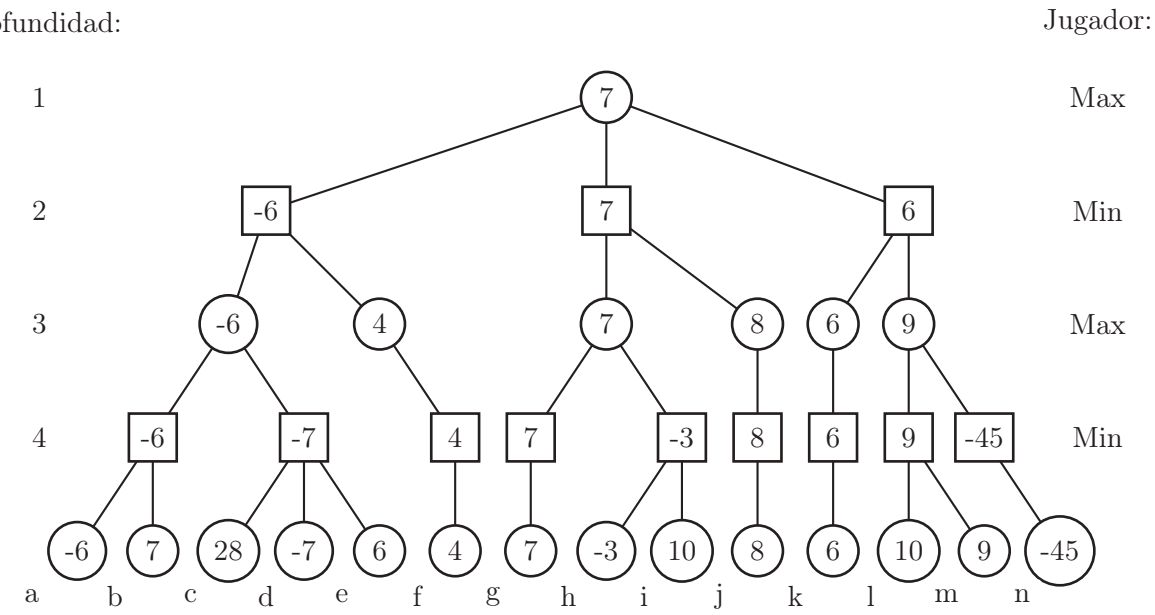

Figura 3.1: Árbol de Búsqueda *Minimax*

éste simplemente escogerá la acción que lo lleve al nodo a ya que es el mínimo entre los dos. De manera contraria, en un nivel de maximización, el jugador Max seleccionará las acciones que lo lleven a maximizar sus ganancias. El resultado de estas operación es un camino desde la raíz del árbol hacia el valor máximo encontrado, para este ejemplo es el camino que lleva al nodo g.

La manera más común de implementar el *Minimax* en juegos de suma cero con acciones alternadas entre jugadores, es a través del algoritmo *Negamax*[33]. La idea del *Negamax* se basa en la equivalencia matemática

$$\text{mín}\big(\text{máx}(x_1,\ldots,x_n),\text{máx}(y_1,\ldots,y_n)\big) = -\,\text{máx}\big(\text{mín}(-x_1,\ldots,-x_n),\text{mín}(-y_1,\ldots,-y_n)\big),$$

donde, $x_1,\ldots,x_n$ son los valores de utilidad de los sucesores de un nodo Max, M_x, mientras que $y_1,\ldots,y_n$ son los valores de utilidad de los sucesores de un nodo Max M_y. El cálculo final es el valor de utilidad de un nodo Min, cuyos sucesores son M_x y M_y. Como puede observarse, las operaciones de un nodo Min son equivalentes a las de un nodo Max pero con signos opuestos.

Esto permite que en vez de tener una alternancia entre dos rutinas, una para calcular el mínimo y otra para calcular el máximo, los valores sólo tiene que ser negados de un nivel a otro y tomar el máximo en cada nivel hará la misma tarea. El pseudocódigo de *Negamax* se muestra en el Algoritmo 1.

Algoritmo 1 NegaMax(d)

Entrada: $d \in \mathbb{N}$ profundidad de búsqueda. ▷ **Variable global** $n \leftarrow$ posición del tablero
Salida: Valor minimax de la posición actual de n ▷ Iniciar con `NegaMax( d)`

```
1: si n es terminal o d = 0 entonces
2:     regresar Evaluación de n
3: fin si
4: Movimientos ← Generar movimientos de n
5: Max ← inf
6: para todo Movimiento ∈ Movimientos hacer
7:     n ← Hacer Movimiento
8:     valor ← −Negamax(d − 1)
9:     n ← Deshacer Movimiento
10:    si valor > Max entonces
11:        Max ← valor
12:    fin si
13: fin para
14: regresar Max
```

La complejidad de este algoritmo es la misma que al algoritmo básico *Minimax*, pero permite representaciones e implementaciones más sencillas. Esto es importante ya que las heurísticas, mejoras y métodos que se presentan más adelante en este trabajo, tendrían que adaptarse para cada rutina de maximización y minimización.

3.3. Algoritmo de Poda α–β

La idea básica del algoritmo α–β [3] es podar partes irrelevantes del árbol que no tienen influencia en el valor que se obtiene en una posición al aplicar el algoritmo *Minimax*. El algoritmo α–β ahorra este tiempo de búsqueda asumiendo la siguiente idea: Si se sabe que cierto movimiento es peor que el mejor encontrado en ese momento, no es necesario dedicar tiempo calculando qué tan malo es. Por lo tanto, se puede pasar por alto dicho movimientos y seguir con el siguiente para ver si es mejor que el que tenemos. Para hacer esto el algoritmo usa un límite inferior y uno superior a los que llamamos α y β respectivamente. Por ejemplo, considérese la operación *Minimax*:

$$\text{máx}\big(\text{mín}(2,4,5), \text{mín}(1,x,y), \text{mín}(10,2,z)\big) = 2,$$

en la cuál es posible establecer el valor *Minimax* de 2, sin necesidad de conocer los valores de x, y y z. Es de notarse que dichos valores son a su vez calculados por una operación *Minimax*, y que cada variable que no necesita ser calculada implica un ahorro de tiempo exponencial a

la profundidad de búsqueda. Cada valor x, y o z se conocen como *ramas* , y al hecho de evitar explorar una de ellas se conoce como *poda* o *corte*.

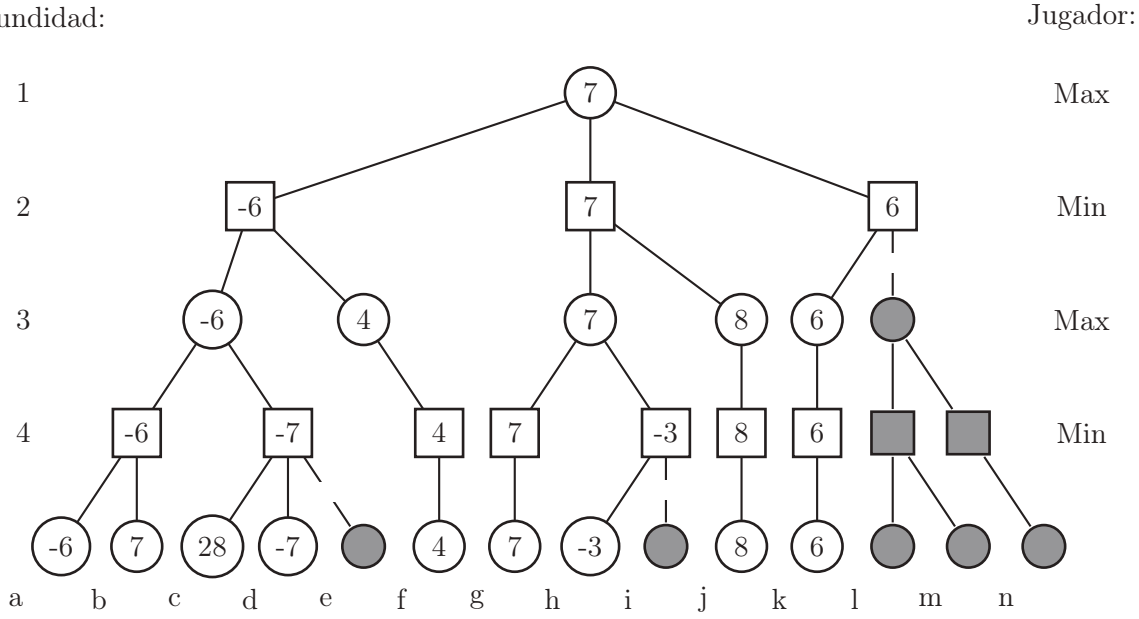

Figura 3.2: Árbol de Búsqueda $\alpha - \beta$.

Las ramas que serán podadas son las que se encuentren fuera de los límites α y β, y al espacio comprendido por estos dos límites se le conoce como *ventana de búsqueda*. Ésta es la responsable de guiar al algoritmo a la solución buscada, ya que suponemos que el valor *Minimax* se encuentra dentro de esta ventana. Los valores de α y β son inicializados de manera a obtener el rango más grande posible entre estos límites, que en el caso del ajedrez serían $-Mate$ para el límite inferior y $Mate$ para el superior. Mientras transcurre la búsqueda esta ventana va disminuyendo su tamaño. El modo en que se modifican estos límites es el siguiente: Cada vez que un valor máximo es encontrado y además es menor que el valor del límite superior β, éste es actualizado, mientras que para el límite inferior α, es actualizado con el mínimo mas grande.

Los cortes ocurren por las siguientes condiciones: en un nivel de minimización, los sucesores con un valor más grande que el límite superior serán cortados, en un nivel de maximización, son los sucesores con un valor más pequeño que el límite inferior. De esta manera, la poda de estos sub-árboles completos deja el valor *Minimax* intacto. Esto aporta un ahorro de tiempo significativo, obteniendo exactamente el mismo resultado que utilizando el algoritmo *Minimax* [3]. Un ejemplo de una poda se puede apreciar en el árbol de la Figura 3.2. Como se observa, no es necesario seguir explorando los nodos que se encuentran después del nodo d, ya el valor

de β en ese momento es de -6 y al explorar el nodo d obtenemos un valor de -7 que es menor a este límite y ocasiona un corte.

La eficiencia de este algoritmo depende completamente del orden en que se exploran los nodos y éste, depende de la ordenación correcta de sus movimientos. En el capítulo 4, se presentan diferentes heurísticas de ordenamiento. El pseudocódigo del algoritmo α–β se presenta en el Algoritmo 2.

Algoritmo 2 BúsquedaAlphaBeta(α, β, d)

Entrada: $d \in \mathbb{N}$ profundidad de búsqueda.
Entrada: $\alpha, \beta \in$ [`-inf`, `inf`] ▷ **Variable global** $n \leftarrow$ posición del tablero
Salida: Valor minimax de la posición actual de n ▷ Iniciar con `BúsquedaAlphaBeta( -inf, inf,`d`)`

```
1: si n es terminal o d = 0 entonces
2:     regresar Evaluación de n
3: fin si
4: Movimientos ← Generar movimientos de n
5: Ordenar Movimientos de acuerdo a una estrategia establecida
6: para todo Movimiento ∈ Movimientos hacer
7:     n ← Hacer Movimiento
8:     valor ← −BúsquedaAlphaBeta( − β, −α, d − 1 )
9:     n ← Deshacer Movimiento
10:    si valor ≥ β entonces
11:        regresar β
12:    fin si
13:    si valor > α entonces
14:        α ← valor
15:    fin si
16: fin para
17: regresar α.
```

3.4. Profundidad iterativa

Los Algoritmos 1 y 2 son de búsqueda tipo *primero profundidad*. Esto quiere decir que exploran el espacio de búsqueda hasta que se alcance la profundidad deseada, antes de explorar otro posible movimiento. Esto representa un problema para los programas de ajedrez, ya que normalmente se requiere que realicen decisiones en tiempo real durante el juego, o están sujetos a restricciones de tiempo para determinar el movimiento que realizarán en cada paso. El problema está en que es imposible determinar el tiempo que tardará una búsqueda, en especial si el programa de ajedrez no se ejecuta bajo un sistema operativo de tiempo real. Si una restricción de tiempo se cumple, y la búsqueda no ha terminado, el programa no sabrá

que acción elegir, es por ello que es necesario asegurar que el motor de ajedrez siempre tenga una acción a tomar.

La profundidad iterativa [14] es una estrategia que pretende evitar este problema. Para ello, llama repetidamente a una rutina de búsqueda aumentando la profundidad d, hasta llegar al valor deseado d_{max}. Para ello se comienza realizando una búsqueda a profundidad $1, 2, \ldots, d_{max} - 1,\ d_{max}$. Al realizarlas de manera incremental siempre tendremos una estimación de un buen movimiento para una profundidad previa, y en caso de que se termine el tiempo siempre tendremos una acción a realizar [23][43].

En principio parece mucho el tiempo de cálculo que invierte la profundidad iterativa en realizar *todas* las búsquedas posibles a una profundidad menor a d_{max}, sólo para poder satisfacer las restricciones de tiempo. Sin embargo, el problema de búsqueda es de complejidad exponencial, y la cantidad de nodos que es necesario expandir en el último nivel siempre será mayor a la cantidad de nodos de *todos* los niveles anteriores [34], por lo que el costo computacional de la profundidad iterativa es despreciable a grandes profundidades de búsqueda. Por otra parte, el uso de la profundidad iterativa puede ser aprovechado para obtener las siguientes ventajas:

- Una búsqueda a profundidad $d-1$ puede darnos una buena idea de cual es la variación principal para una búsqueda a una profundidad d. Esto lleva a intentar primero estos movimientos, ya que estos son generalmente buenos, lo que se traduce en un mejor ordenamiento de los movimientos. El desempeño del algoritmo α–β mejorará notablemente al depender éste de la manera en que son ordenados los movimientos.
- El valor *Minimax* que se obtiene de una búsqueda a profundidad $d-1$, puede ser usado como el centro para una ventana de búsqueda de aspiración α–β (*aspiration search*) para una búsqueda a profundidad d, ya que es probable que el valor *Minimax* a nivel superior sea muy similar al valor calculado. Este método se desarrollará en el capítulo siguiente.
- Si a una profundidad menor a d_{max} se encuentra una solución con valor máximo (esto es, un valor de $Mate$), no será necesario explorar hasta la profundidad d_{max}.
- Las heurísticas dinámicas y las tablas de transposición, las cuales se presentan en el capítulo siguiente, pueden ser actualizadas con información valiosa en cada búsqueda a menor profundidad. Esta es la principal y verdadera ventaja que se obtiene mediante la profundidad iterativa. Si se actualizan correctamente los valores de las tablas de transposición y de las heurísticas dinámicas en una búsqueda en profundidad $d-1$, éstas tienden a llevar la búsqueda a profundidad d por líneas que son suficientemente buenas

para provocar la mayor cantidad posible de cortes, lo que se traduce en un ahorro de tiempo considerable.

El poder contar con una mejor ordenación de lo nodos iniciales, junto con una buena estimación de la ventana de búsqueda α–β que se logra con el uso de la profundidad iterativa, se traduce en un ahorro de tiempo de cómputo en la búsqueda mucho mayor al tiempo extra invertido al explorar a profundidades menores a la deseada. En general, en forma práctica, la suma de todos los nodos generados por las iteraciones de 1 hasta $d-1$ es mucho menor que el número de nodos generados a la profundidad d. Esto quiere decir que el mayor tiempo de cómputo será utilizado en este nivel, por lo que invertir tiempo en niveles inferiores para poder reducir el número de nodos explorados en este nivel es aceptable [21]. Por todas estas razones, la profundidad iterativa es reconocida como un componente fundamental en cualquier programa de ajedrez. El pseudocódigo de profundidad iterativa se muestra en el Algoritmo 3.

Algoritmo 3 Profundidad Iterativa

Entrada: $d \in \mathbb{N}$ profundidad de búsqueda.
Entrada: $\alpha, \beta \in$ [`-inf`, `inf`]
1: **para** $d = 1, 2, \ldots, d_{max}$ **hacer**
2: $valor \leftarrow -$`AlphaBeta(` $-\alpha, -\beta, d$ `)`
3: **si** $valor == Mate \,||\, valor == -Mate$ **entonces**
4: break
5: **fin si**
6: **fin para**

3.5. Búsqueda Quiescence y el efecto horizonte

La búsqueda *Quiescence* [24] es un algoritmo usado para evaluar árboles de juego *Minimax*, debido a un problema irresoluble que sufren todos los algoritmos de búsqueda en árboles de juego, llamado *efecto horizonte*. Este problema ocurre debido a que los algoritmos usados sólo exploran hasta una profundidad límite dada.

El efecto se presenta cuando una posición de tablero perdedora se encuentra más allá de la profundidad máxima de búsqueda, por lo que no podrá ser vista [4]. El programa procederá entonces a realizar un movimiento, el cual si se pudiera explorar todo el árbol resultaría ser un movimiento perdedor, pero la función de evaluación devolverá un valor *Minimax* favorable a ciertas profundidades de búsqueda. Movimientos más tarde, el algoritmo de búsqueda será capaz de explorar a la profundidad necesaria para que la función de evaluación retorne el

valor correcto e indicará que la posición es perdedora. Desafortunadamente, en este punto, el programa ya no es capaz de evitar la posición perdedora.

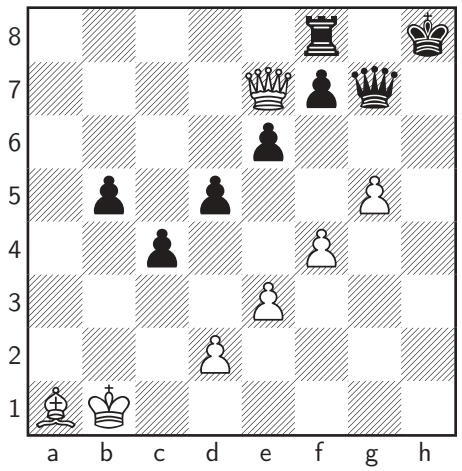

Figura 3.3: Ejemplo del efecto horizonte.

El efecto horizonte se ilustra en la Figura 3.3. En el turno del jugador negro, la búsqueda, a profundidad d ha terminado y estableció que la posición mostrada en la Figura 3.3 tiene un valor positivo, debido a la superioridad en material de las negras, lo que indicará al algoritmo de búsqueda que ésta es una posición favorable. En realidad, la posición es muy mala, debido a que se está perdiendo a la reina que se encuentra en la línea de ataque del alfil enemigo y no puede moverse debido a que su rey se encuentra detrás. Para poder haber evitado esta posición y que la función de evaluación encontrara que el valor real de la posición, era necesario explorar 10 jugadas más a futuro. La búsqueda se extiende debido a que se pueden interponer peones en la línea de ataque del alfil, pero lo único que se logra es posponer la pérdida de la reina. La posición para el jugador negro debería estimarse cercana a $-Mate$, aunque el jugador negro tenga una torre en vez de un alfil y más peones en esa posición. Para mitigar el efecto horizonte se propuso el uso de la búsqueda *Quiescence*.

Los jugadores humanos usualmente tienen suficiente intuición para decidir si debe abandonar líneas de movimientos que parezcan malas, o buscar movimientos prometedores a gran profundidad. La búsqueda Quiescence intenta emular este comportamiento al seleccionar posiciones *relevantes* o *inestables* para buscar a mayor profundidad. La heurística pretende evitar trampas escondidas y obtener una mejor estimación del valor real de las posiciones. De esta manera, se mitiga el efecto horizonte al continuar la búsqueda, aun cuando la profundidad

límite se haya alcanzado, continuando hasta que no existan más posiciones *inestables*. De ahí su nombre ya que *Quiescence* significa tranquilidad.

Algoritmo 4 Quiescence(α, β)

Entrada: α, β ▷ **Variable global** $n \leftarrow$ posición del tablero
Salida: Valor minimax para el primer estado estable de n ▷ Iniciar con `Quiesce(-`β`,-`α`)`

```
1: valor ← Evaluación de n
2: si valor ≥ β entonces
3:     regresar β
4: fin si
5: si valor < α entonces
6:     α ← valor
7: fin si
8: Movimientos ← Generar movimientos inestables n
9: Ordenar Movimientos de acuerdo a una estrategia establecida
10: para todo Movimiento ∈ Movimientos hacer
11:     n ← Hacer Movimiento
12:     valor ← -Quiescence( −β, −α)
13:     n ← Deshacer Movimiento
14:     si valor ≥ β entonces
15:         regresar β
16:     fin si
17:     si valor > α entonces
18:         α ← valor
19:     fin si
20: fin para
21: regresar α
```

No todas las posiciones a evaluar presentan el efecto horizonte o requieren de la búsqueda *Quiescence*. Es por ello que solamente es necesario aplicar la búsqueda *Quiescence* a ciertas posiciones que provocan grandes cambios en la función de evaluación. Normalmente se consideran posiciones que contengan movimientos de captura y los movimientos de jaque [15]. Idealmente, también se debe considerar los peones pasados, aquellos que están cerca de ser promocionados, y posiciones selectas de jaque [22].

A pesar de los obvios beneficios de estas ideas, el campo de la búsqueda *Quiescence* no es claro [18], ya que no hay teoría acerca de la selección de qué movimientos deben ser tomados en cuenta para continuar la búsqueda. Aunque una posición pueda parecer candidata para la búsqueda *Quiescence*, puede no serlo, o peor aún, no parecerlo. Los algoritmos actuales son atractivos ya que son simples, pero desde el punto de vista de un jugador de ajedrez, dejan mucho que desear, especialmente cuando se tratan de movimientos de bifurcación y amenazas de jaque. Aunque las aproximaciones actuales son razonablemente efectivas, se necesita méto-

dos más sofisticados para mejorar la búsqueda, o para identificar movimientos inestables [24]. Aún con el uso de esta extensión de búsqueda no se elimina el efecto horizonte, sólo se mitiga. El algoritmo para implementar la búsqueda *Quiescence* se muestra en el Algoritmo 4.

3.6. Extensiones de búsqueda

Las extensiones de búsqueda son esenciales para cualquier programa de ajedrez con un buen poder de juego [9]. La idea de las extensiones es simple: se intenta invertir más tiempo de búsqueda en aquellos movimientos que luzcan «interesantes» y valgan la pena analizarlos más cuidadosamente. Esto intenta emular la forma en que los humanos juegan ajedrez, ya que son excelentes descartando líneas de búsqueda y se enfocan en ciertas líneas interesantes de juego.

En una búsqueda de árbol estándar, el método de búsqueda es llamado recursivamente hasta que se alcanza el límite de búsqueda máximo y después se hace uso de la búsqueda *Quiscence*. Sin embargo, para ciertos nodos, la búsqueda *Quiscence* no podrá encontrar una buena estimación, por lo que sería mejor extender la búsqueda principal un poco más para observar si estos movimientos se convierten en algo interesante [1].

A continuación, se enlistan las extensiones que han sido implementadas en el motor de ajedrez *Buhochess*:

Extensiones por jaque. Si el lado que le toca jugar se encuentra en jaque se extiende la búsqueda a una profundidad mayor. Esta extensión incrementó el poder de juego del programa dramáticamente.

Extensiones por un movimiento. A veces ocurre que sólo se tiene un movimiento legal por hacer, normalmente se debe a que nuestro rey se encuentra en peligro. El costo extra por explorar estos nodos es mínimo debido a que el factor de ramificación es solamente de 1.

Extensión por amenazas. Existen técnicas como la poda de movimiento nulo (presentada en el capítulo 5), donde podemos determinar que existe una amenaza en futuros movimientos pero no sabemos de qué se tratan o qué tan graves son. Si sabemos que están por ocurrir, vale la pena extender la búsqueda para tratar encontrar la forma de evitar estas amenazas.

Extensiones por peones. Si existe un peón que está a punto de coronar, o que coronó, vale

la pena expandir la búsqueda ya que se trata de una jugada «interesante» debido a que normalmente estas jugadas suponen una gran ventaja o desventaja para uno de los jugadores.

Seguridad del rey. Si la seguridad del rey decae dramáticamente, esta posición requiere que sea analizada más cuidadosamente, por lo que se expande la búsqueda. Por ejemplo, nuestro rey puede quedar desprotegido por un sacrificio posicional.

Al hacer uso de las extensiones de búsqueda se debe tener tener en mente que estas búsquedas deben terminar en algún momento sin importar qué tanto más se quiera extender la búsqueda en el árbol. El programa *BuhoChess* hace uso de dos límites para las profundidades. El primero límite está dado por la profundidad iterativa e indica que la búsqueda principal terminó. El segundo límite es usado para indicar la profundidad máxima que puede extenderse la búsqueda, en el motor *Buhochess*, este valor es 32. Al alcanzar el segundo límite se hace un llamado a la función de evaluación estática, o bien podría realizarse una búsqueda *Quiscence*. Debe tenerse cuidado al implementar las extensiones de búsqueda y tener en cuenta cuándo y cuáles extensiones se usarán, ya que se podría incrementar el número de nodos explorados significativamente lo que disminuiría el rendimiento del programa de ajedrez [9].

3.7. Comentarios finales

En este capítulo se presentó el algoritmo básico de búsqueda *Minimax*, que permite explorar árboles de juego y encontrar el valor óptimo del mismo. Sin embargo, éste debe explorar todo el árbol de juego a cierta profundidad, lo que es sumamente ineficiente y poco práctico para árboles tan grandes como el del ajedrez. Es por ello que se desarrollaron mejoras para este algoritmo como la poda α–β presentada en el capítulo, que genera el mismo resultado que *Minimax*, pero permite ahorros de tiempo considerables, al podar partes del árbol que resultan irrelevantes.

Debido a la imposibilidad de predecir el tiempo total utilizado en una búsqueda y a las restricciones de tiempo a las que está sujeto el ajedrez para realizar un movimiento, es necesario contar con una acción que tomar si el tiempo termina antes que la búsqueda retorne el mejor valor encontrado por ésta. La profundidad iterativa es usado con este propósito, además que ofrece otras ventajas que son explicadas en el capítulo 4.

Las extensiones de búsqueda y la búsqueda *Quiscence* permiten una mejor evaluación de los nodos explorados, y ayudan a mitigar un problema que presentan y sufren los algoritmos

de búsqueda utilizados, llamado efecto horizonte, que puede provocar grandes errores en la búsqueda. Este efecto ocurre debido a la imposibilidad de explorar todo el árbol de juego.

Capítulo 4

Mejoras a la búsqueda

4.1. Introducción

El algoritmo α–β devuelve *exactamente* el mismo resultado que el algoritmo *Minimax*, sin embargo, el número de nodos visitados puede ser bastante diferente, qué tan bien lo haga depende del orden en que los movimientos son explorados. En el peor caso, la poda α–β no genera ningún corte, debido a que los nodos son explorados en el peor orden posible. Así el algoritmo $\alpha-\beta$ se convierte un algoritmo *Minimax*, con una complejidad de $\mathcal{O}(\omega^d)$. En el mejor caso, la ordenación de los movimientos es perfecta, los mejores movimientos son seleccionados primero en cada nodo por lo que el número de cortes es máximo. En [38] se demostró que la complejidad del mejor caso depende del hecho de que d sea par o impar. Para d par, la complejidad temporal será $\mathcal{O}(2\omega^{d/2}+1)$ y para d impar será $\mathcal{O}(\omega^{(d+1)/2}+\omega^{(d-1)/2}-1)$. Por ejemplo una búsqueda a profundidad $d=6$, asumiendo $\omega=20$ explorará $20^6=64,000,000$ nodos en el peór de los casos, mientras que en el mejor, explorará solamente $2\cdot 20^3+1=16,001$ nodos.

Un adecuado orden de los movimientos es entonces crucial para poder provocar más cortes y así lograr una búsqueda α–β más efectiva, una ligera mejora en el ordenamiento de éstos puede aumentar el rendimiento de la búsqueda alrededor de 60 % al 100 % [21]. Esta dependencia de la ordenación de los movimientos para un buen desempeño de los algoritmos ha llevado al desarrollo de heurísticas de ordenación de movimientos, así como modificaciones al algoritmo α–β que permiten explorar solamente alrededor de 15 % a 30 % más nodos que el árbol de búsqueda mínimo. Estas heurísticas se presentan en la siguiente sección y usualmente son independientes del dominio por que pueden ser usados en otros tipos de problemas.

A pesar de que el algoritmo α–β es extremadamente efectivo en comparación al algoritmo $Minimax$ puro, la complejidad del árbol de juego es de tipo exponencial por lo que aún en el mejor de los casos, la cantidad de nodos será demasiada ya que sigue siendo exponencial. Por este motivo, se ha invertido mucho esfuerzo para mejorar el rendimiento del algoritmo usando diversos métodos, de los cuales se tratarán en este capítulo (además de las heurísticas de ordenación), la búsqueda de aspiración, la búsqueda de ventana mínima y las tablas de transposición.

4.2. Heurísticas estáticas

Las heurísticas de ordenación estáticas no dependen de la información de búsquedas previas, solo requieren información que obtienen de la posición actual de tablero. Existen muchas heurísticas independientes del dominio, que funcionan bien en juegos como ajedrez como la heurística MVV/LVA. Ésta, es usada para ordenar los movimientos de captura de una manera razonable. Las capturas generalmente ocasionan grandes cambios en la función de evaluación, porque o son movimientos muy buenos (por ejemplo, ganar la reina enemiga) o muy malos (por ejemplo, perder nuestra reina). Las de capturas son ordenadas por la relación que existe entre víctima mas valiosa *(Most Value Victim, MVV)*, y el agresor menos valioso *(Less Value Agresor, LVA)*.

Para aplicar la heurística, primero se busca a la víctima más valiosa entre todas las piezas enemigas atacadas. Después, se procede a encontrar al atacante menos valioso. Lo que se busca es ganar una gran ventaja material sin sacrificar mucho material del propio. Se ordenarán los movimientos en relación a la pieza atacante y a la atacada. Si tenemos un peón que ataca a una reina enemiga, este movimiento es considerado muy bueno de acuerdo a la heurística MVV/LVA, por lo que se ordenará primero. En último lugar se ordena el movimiento de un peón atacado por la Reina, ya que si éste está defendido se perderá mucho material.

Las capturas es una de las partes más importantes en la ordenación de los movimientos. Normalmente si se gana material se considerarán primero que las demás heurísticas de ordenación. Sin embargo, los movimientos donde se pierde material pueden considerarse hasta en último lugar de todas las demás heurísticas. La manera que se implementó MVV/LVA en el motor *BuhoChess* es:

`Valor` ← `Constante + 100*Valor_de_Víctima - 10*Valor_de_Agresor`.

La constante `Constante` es utilizada para diferenciar el valor dado por esta heurística, respecto a otras. La constante podría tener un valor de 1,000,000 que sería mucho mayor que cualquier

valor generado por las demás heurísticas de ordenamiento. Así el movimiento será considerado antes que cualquier otro. Si se trata de una captura perdedor la constante podría tener otro valor para colocarla en otro rango deseado. Es importante resaltar que se calcula el valor de cada posición para todas las heurísticas y de acuerdo a sus valores acumulados, se realiza la ordenación de los movimientos.

En el caso del motor de ajedrez *Buhochess*, los movimientos que pierden material están posicionados después que los movimientos asesinos (los cuales se verán más adelante), pero antes que los posicionales y demás tipos de movimiento. El valor de la víctima se multiplica por 100 para asegurar que, al restarle el valor del agresor, nunca sean valores negativos.

4.3. Heurísticas dinámicas

Las heurísticas de ordenación de movimientos dinámicas recaudan información acerca de las jugadas durante las búsquedas anteriores y la utilizan para un mejor ordenamiento de los movimientos. Ejemplos de éstas, son: la heurística de historia [35] y la heurística asesina [36]. Ambas heurísticas son independientes del dominio y funcionan almacenado información sobre los movimientos que han mostrado buenos resultados de previas búsquedas o posiciones previamente exploradas.

4.3.1. Heurística de historia

La heurística de historia[35] es una forma simple y poco costosa de ordenar los movimientos dinámicamente en los nodos interiores del árbol de juego. La idea básica es que la realización de un movimiento lleva a posiciones sucesoras que tiene muchas características similares a su predecesor. Es muy probable que varias de las características importantes de una posición (las que se utilizan en la función de evaluación), no cambien significativamente después de un movimiento. Si un movimiento es considerado bueno en una posición, se asume que será bueno para una posición similar, solo difiriendo en la posición de una pieza. Esta diferencia menor puede no haber cambiado la posición lo suficiente como para alterar las características significativas que hacen al movimiento el mejor para la posición original.

Para la heurística de historia, es necesario almacenar un historial de éxitos de cada movimiento que halla sido seleccionado en otras búsquedas. El movimiento que más se tenga seleccionado en posiciones similares, puede ser considerado como el primero que se explorará entre los movimientos posicionales, para dicha posición. Un movimiento *suficientemente* bueno

puede definirse como

- El Movimiento cuyo valor es mayor a β, provocando un corte.
- El Movimiento que aumente el valor de *Minimax*.

Debe aclararse que un movimiento *suficientemente* bueno no implica forzosamente que sea el mejor posible. Un movimiento que causa un corte indica que simplemente fue lo suficiente bueno para causar que la búsqueda termine en ese nodo, pero no hay garantía de que si la búsqueda hubiera continuado, uno mejor, aunque irrelevante para la búsqueda anterior, pudiera ser encontrado.

Cada vez que un movimiento *suficientemente* bueno es encontrado, la puntuación de historia asociada a éste es aumentado. Así, movimientos que son considerado buenos alcanzan rápidamente altas puntuaciones. Los movimientos son ordenados de acuerdo a su puntuación de historia de éxito previo. La heurística está basado en la experiencia, ya que se va adquiriendo información sobre lo que puede ser de utilidad en el futuro. Así, la información de búsquedas previas es acumulada y distribuida a través de todo el árbol.

La implementación de esta heurística se puede apreciar en el Algoritmo 5. Para este algoritmo, es necesario guardar una lista de movimientos, asociando el índice de éxitos por cada uno. En el ajedrez, esto se logra utilizando tablas *hash*, donde la información almacenada es de solamente dos números que indican la casilla origen y la destino. Aunque esta representación simplista pierda muchos de los detalles del movimiento (la pieza que se esta moviendo por ejemplo), permite representaciones con tablas de 4,096 entradas (64x64). Incluir más información de los movimientos no parece incrementar su efectividad de acuerdo a la experiencia con el desarrollo del motor *Buhochess*. Si se lleva este almacenamiento de información al extremo, el resultado sería una tabla de transposición, la cual se decribe más adelante en este mismo capítulo.

Para asignar el peso de un movimiento *suficientemente* bueno, existen dos consideraciones: por un lado, entre más profundo sea el sub-árbol explorado, mas seguro es el valor *Minimax*. Por otra parte, entre más profundo sea el árbol de búsqueda, mayor será la diferencia entre dos posiciones arbitrarias en el árbol de búsqueda y menos características tendrán en común. Por lo tanto, movimientos *suficientes* cercanos a la raíz del árbol tienen más potencial de ser útiles a través del árbol que aquellos movimientos *suficientes* que son cercanos a las hojas. El peso que se usa para otorgar un valor de historia normalmente está dado por 2^d, donde d es la profundidad del sub-arbol explorado. Este esquema da más peso a movimientos que se

encuentras más cercanos a la raíz que aquellos cercanos a las hojas del árbol.

Algoritmo 5 Heuristica de Historia

Entrada: $d \in \mathbb{N}$ profundidad de búsqueda.
Entrada: $\alpha, \beta \in$ [`-inf`, `inf`] ▷ **Variables globals** $n \leftarrow$ posición del tablero, `valor` $\leftarrow$ tabla de valores
Salida: Valor minimax de la posición actual de n. ▷ Iniciar con `BúsquedaAlphaBeta( -inf, inf,`d`)`

si n es terminal **o** $d = 0$ **entonces**
  regresar Evaluación de n.
fin si
$Movimientos \leftarrow$ Generar movimientos de n.
Ordenar $Movimientos$ de acuerdo a `valor[`$Movimientos$`]`
para todo $Movimiento \in Movimientos$ **hacer**
  $n \leftarrow$ Hacer $Movimiento$
  $valor \leftarrow -$`BúsquedaAlphaBeta(` $-\beta, -\alpha, d-1$ `)`
  $n \leftarrow$ Deshacer $Movimiento$
  si $valor \geq \beta$ **entonces**
    `valor[`$Movimiento$`]`$\leftarrow$`valor[`$Movimiento$`]`$+2^d$
    regresar β
  fin si
  si $valor > \alpha$ **entonces**
    `valor[`$Movimiento$`]`$\leftarrow$`valor[`$Movimiento$`]`$+2^d$
    $\alpha \leftarrow valor$
  fin si
fin para
regresar α

4.3.2. Heurística asesina

A menudo, durante la búsqueda la mayoría de los movimientos pueden ser refutados rápidamente, usualmente usando el mismo movimiento. La heurística asesina [36] recuerda para cada nivel de profundidad de búsqueda los movimientos que parezcan que están causando la mayoría de los cortes, de allí el nombre de asesina. La idea detrás de esta heurística es que dada una posición a una profundidad específica, las posiciones que se encuentran a la misma profundidad pueden cambiar muy poco, por lo que el movimiento que se encontró como *asesino* en dicha posición puede ser considerado *asesino* en todas las posiciones a la misma profundidad, siempre y cuando el movimiento sea legal. Lo que se espera es que, en ambas posiciones, el movimiento pueda ocasionar un corte.

Esta heurística es una generalización de la heurística de historia pero a diferencia de ella, la heurística asesina sólo mantiene información de pocos movimientos en cada profundidad del árbol. Selim y Monroe [36] observaron mejores resultados si solamente se mantenían 2

posiciones por profundidad en el árbol. La información de los movimientos no se comparte a través de todo el árbol de búsqueda, sólo comparte información a través de las posiciones que se encuentren a la misma profundidad.

Un ejemplo de una posición donde la heurística asesina es efectiva es mostrada en la Figura 4.1. En la posición mostrada, después de cualquier movimiento del jugador negro, el jugador blanco encuentra que el movimiento *Nc7* es un movimiento asesino, ya que da mucha ventaja y por ende es un movimiento que normalmente el jugador negro no permitiría. Al probar primero este movimiento asesino, es muy probable que se genere un corte. Aunque la *heurística asesina* ofrece grandes ventajas, ésta no es tan efectiva en todas las posiciones del ajedrez. Normalmente, su uso puede provocar una reducción del tamaño del árbol de búsqueda alrededor de 10 % a 20 % [36].

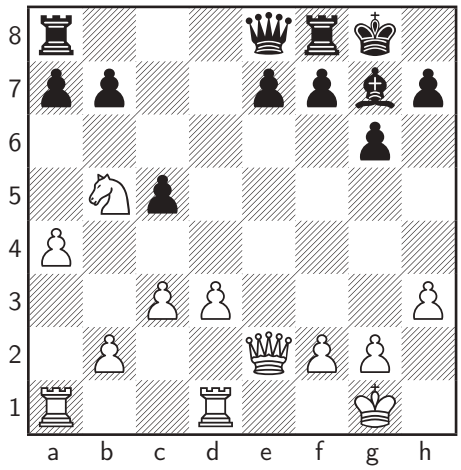

Figura 4.1: Ejemplo de un movimiento asesino Nc7.

Esta heurística puede ser implementada de diversas formas, la más popular y la que mejores resultados ha mostrado es mantener dos movimientos asesinos por nivel de la profundidad y usar tablas que son externas a la búsqueda para mantener estos movimientos. Al igual que la *heurística de historia* para cada uno de estos movimientos un contador es conservado, y cada vez que un corte ocurre este contador es incrementado, normalmente con un peso de 1. En caso de que un movimiento ocasione un corte y no sea alguno de estos dos movimientos, éste remplazará al movimientos asesino con el contador más bajo. Cuando se encuentre una posición en la búsqueda para la cual se tiene un movimiento asesino, a éste se le asigna un valor más alto para que pueda ser ordenado antes que los otros movimientos, esperando que ocasione un corte.

4.4. Búsqueda de aspiración

La idea en que se basa la búsqueda de aspiración [40] es mejorar la ventana de búsqueda (el intervalo $[\alpha, \beta]$), aprovechando el uso de la profundidad iterativa. Al realizar una búsqueda con el algoritmo α–β, mientras más pequeña sea la ventana de búsqueda, más cortes se generarán. Sin embargo, si el valor *Minimax* se encuentra fuera del intervalo $[\alpha, \beta]$ se tendrá que recomenzar la búsqueda . En general, en los algoritmos anteriores, al asumir que no se conoce el valor *Minimax* exacto, se comienza con una ventana inicial de búsqueda $[-Mate, Mate]$, que pueda englobar cualquier valor devuelto por el algoritmo. Si pudiéramos hacer una estimación del valor que estamos buscando, podríamos ajustar los límites de esta ventana de búsqueda, ya que entre más estrecha sea, más rápido será el desempeño del algoritmo[29][30].

En lugar de usar una ventana inicial de $[-Mate, Mate]$, se utilizará una ventana $[V - \epsilon, V + \epsilon]$, donde V es una estimación del valor y ϵ es el error esperado de V respecto al valor *Minimax*. El valor V se estima a partir de los resultados de la búsqueda a una profundidad menor, al utilizar la profundidad iterativa. Si el valor *Minimax* no se encuentra en la ventana de búsqueda, se debe repetir la búsqueda con valores creibles para los límites. Típicamente estas fallas ocurren cuando hay gran diferencia en los valores de la función de evaluación, como ganar o perder material. En estos casos el tiempo extra para volver a revisar una posición es aceptable. Los tres posibles casos que pueden ocurrir al hacer uso de la búsqueda de aspiración son:

- El valor retornado se encuentra entre los límites α y β: en este caso, el valor del árbol de juego ha sido determinado sin mayor problema y no se necesita realizar futuras búsquedas.
- El valor retornado es menor o igual a α: el valor real del árbol de juego no pudo ser determinado, lo único que pudimos determinar es que el valor real es menor o igual a α. La búsqueda debe volver a realizarse para poder determinar el valor real utilizando una ventana de búsqueda $[-Mate, V]$.
- El valor retornado es mayor o igual a β: ell valor real del árbol de juego no pudo ser determinado, lo único que pudimos determinar es que el valor real es mayor o igual a β. La búsqueda debe volver a realizarse usando la ventana $[V, Mate]$.

Se ha experimentado el uso de «ventanas corredizas» de la forma $[V, V+$valor de una pieza$]$ para las búsquedas cuando falla la búsqueda, en lugar de $[V, Mate]$. Este método es regularmente efectivo y ofrece mejores resultados ya que se sigue evitando usar valores muy grandes para

la ventana, sin embargo pueden llevar a excesivas búsquedas innecesarias. Esto normalmente ocurre cuando una posición con valor de $Mate$ o $-Mate$ se encuentra cercana a la posición actual, o cuando la cantidad de material que está siendo ganada o perdida es importante.

La búsqueda de aspiración es una mejora delicada del algoritmo α–β [40]. Si se utiliza un ventana muy grande muy pocos o ningún corte ocurrirá durante la búsqueda. Si se usa una ventana muy pequeña, el valor buscado no se encontrará dentro de la ventana de búsqueda, y tendremos que hacer excesivas búsquedas con los valores correctos para poder determinar el valor real, lo que se traduce en un desempeñó peor que no hacer uso de la búsqueda de aspiración. Por lo observado en *BuhoChess*, el tamaño de esta ventana depende en gran medida del número de características que se toman en cuenta en la función de evaluación, ya que si son pocos los valores no serán muy distintos entre sí y la probabilidad de que ocurra un corte disminuirá. El tamaño recomendado para ajustar estas ventanas es con ϵ igual al valor del peón entre tres o entre cuatro [40]. En *BuhoChess* se utiliza un ϵ del valor del peón entre tres, que fue el que mejores resultados demostró. La manera que se implementa la búsqueda de aspiración se muestra en el Algoritmo 6.

Algoritmo 6 Búsqueda de aspiración

Entrada: $d \in \mathbb{N}$ profundidad de búsqueda.
Entrada: $\alpha, \beta \in$ [`-inf`, `inf`]

para $d = 1, 2, \ldots, d_{max}$ **hacer**
  $\alpha \leftarrow V - \epsilon$
  $\beta \leftarrow V + \epsilon$
  $V \leftarrow$ `BúsquedaAlphaBeta(`α, β, d `)`
  si $V \geq \beta$ **entonces**
    $V \leftarrow$ `BúsquedaAlphaBeta(`V, `inf`, d `)`
  si no
    si $V \leq \alpha$ **entonces**
      $V \leftarrow$ `BúsquedaAlphaBeta(-inf,`V, d `)`
    fin si
  fin si
  si $valor == Mate$ **ó** $valor == -Mate$ **entonces**
    break
  fin si
fin para

4.5. Búsqueda por ventana mínima

Los algoritmos de búsqueda basados en ventanas mínimas [27] [30] [5], han demostrado un mejor rendimiento que el algoritmo α–β normal, cuando se utilizan buenas heurísticas de

ordenación de movimientos y se realiza la búsqueda a profundidades importantes. La idea básica es que es más económico en tiempo comprobar que un movimiento genera un corte a determinar su valor exacto de éste. Con una buena ordenación de los movimientos, los cortes durante la búsqueda normalmente ocurrirán en los primeros movimientos (normalmente el 90 % del tiempo [27]) y rara vez en los demás movimientos.

Si se supone una ordenación de movimientos perfecta, el primer movimiento explorado será siempre el que incremente el valor de α o provoque un corte. Bajo esta suposición se puede utilizar una ventana de búsqueda mínima para comprobar si el resto de los movimientos tienen el potencial de superar a α. Esta búsqueda se hace usando una ventana de búsqueda de $[\alpha, \alpha + 1]$, donde 1 es la unidad mínima de incremento en la función de evaluación. A esto se le conoce como ventana mínima.

La búsqueda de ventana mínima no otorga valores exactos y es considerada como una búsqueda booleana, ya que solamente indica si la suposición sobre la variación principal era correcta. Si la suposición falla, se tiene que volver a realizar la búsqueda con los límites de la ventana reales, para determinar el valor exacto del subárbol.

Se ha mostrado que, para árboles grandes y con heurísticas de ordenación de movimientos eficientes, las técnicas de ventanas mínimas proveen una ventaja con respecto a la poda α–β cercana a un 10 % menos de nodos explorados [27]. Para árboles sin ordenar, se sabe que su complejidad es asintóticamente equivalente a un algoritmo α–β simple, aunque puede llegar a revisar más nodos, ya que si la ordenación falla, entonces se tendrán que volver a revisar los subarboles de esos movimientos. El pseudocódigo para ventana mínima con variación principal se presenta en el Algoritmo 7.

4.6. Tablas de transposición

Una tabla de transposición ([39], [17]) es un componente de suma importancia para cualquier programa de ajedrez ya que actúa como una base de datos donde se almacenan los resultados de búsquedas previamente realizadas. Esto ayuda a reducir drásticamente el espacio de búsqueda con un pequeño impacto negativo, ya que intercambiamos memoria para ganar velocidad de procesamiento. Debido a que el espacio de búsqueda en el ajedrez es en realidad un gráfo acíclico, es posible llegar a una misma posición desde diferentes estados. Por ejemplo, la secuencia de movimientos *e4, e5, Nc6* y la secuencia de movimientos *Nc6, e5, e4* llevan a la misma posición, aunque las secuencias fueron distintas; a ésto, se conoce como *transposición.*

Algoritmo 7 Búsqueda por ventana mínima `BúsquedaPV(`α, β, d`)`

Entrada: $d \in \mathbb{N}$ profundidad de búsqueda
Entrada: $\alpha, \beta \in$ [`-inf`, `inf`] ▷ **Variable global** $n \leftarrow$ posición del tablero
Salida: Valor minimax de la posición actual de n ▷ Iniciar con `BúsquedaPV( -inf, inf,`d`)`

si n es terminal **o** $d = 0$ **entonces**
  regresar Evaluación de n.
fin si
$bSearchPv \leftarrow$ **True**
$Movimientos \leftarrow$ Generar movimientos de n
Ordenar $Movimientos$ de acuerdo a una estrategia establecida
para todo $Movimiento \in Movimientos$ **hacer**
  $n \leftarrow$ Hacer $Movimiento$
  si bSearchPv **entonces**
    $valor \leftarrow -$`BúsquedaPV(`$-\beta, -\alpha, d-1$`)`
  si no
    $valor \leftarrow -$`BúsquedaPV(`$-\alpha-1, -\alpha, d-1$`)`
    si $valor > \alpha$ **entonces**
      $valor \leftarrow -$`BúsquedaPV(`$-\beta, -\alpha, d-1$`)`
      $n \leftarrow$ Deshacer $Movimiento$
    fin si
  fin si
  $n \leftarrow$ Deshacer $Movimiento$
  si $valor \geq \beta$ **entonces**
    regresar β
  fin si
  si $valor > \alpha$ **entonces**
    $\alpha \leftarrow valor$
    $bSearchPv \leftarrow$ **False**
  fin si
fin para
regresar α.

Estas transposiciones ocasionan mucha redundancia en espacio de búsqueda, especialmente en la fase final del juego, por lo que, cuando nos encontremos con una transposición, es beneficiosos poder recordar qué determinamos la última vez que la exploramos. Así, es posible evitar repetir de nuevo toda la búsqueda para dicha posición. Una tabla de transposiciones es implementada mediante una tabla *hash* de gran capacidad [25] en la que se almacenan posiciones previamente exploradas.

La principal ventaja del uso de tablas *hash* es que otorgan tiempos de acceso rápidos y constantes, debido a que éstas se consultarán por cada nodo que exploremos durante la búsqueda. Idealmente, por cada posición en el tablero, debe de haber una entrada en la tabla de

transposición. Sin embargo, una tabla de esas dimensiones sería prohibitiva en requerimientos de memoria, y su aplicación podría ser lenta si, por ejemplo, para el uso de la memoria de la tabla *hash* es necesario hacer consultas al disco duro. Para hacer uso de las tablas hash es necesario poder generar claves (*hash keys*), que nos permitan acceder en la información que hemos almacenado.

4.6.1. Claves Zobrist

Por mucho, el método de *hasheo* más popular es el propuesto por Zobrist [45]. Él observó que una posición de ajedrez está compuesta por 6 tipos de piezas para el jugador blanco y 6 para el negro, los valores que representaban los derechos de enroque de cada jugador, los peones en paso que existan y, por último, el jugador en turno. La idea de Zorbist fue generar la clave a partir de un arreglo tridimensional con índices para cada tipo de pieza, color y la posición de cada una, en el cual cada posición en el arreglo contiene un número aleatorio, el arreglo que propuso tiene la siguiente forma:

$$\texttt{ArregloZobrist[Color][Pieza][Posicion]} \leftarrow \texttt{random},$$

donde `Color` $\in \{0, 1\}$, `Pieza`$\in \{0, \ldots, 5\}$ y `Posición` $\in \{0, \ldots, 63\}$. Para calcular una clave Zobrist se realizan los siguientes pasos.

1. Generar un conjunto de números aleatorios para cada posición en el arreglo tridimensional, se recomiendan número aleatorios entre 0 y 64 bits. También se generan 16 números para las posiciones de los peones en paso (8 para cada jugador), 8 números para los derechos de enroque (4 para cada jugador), y 1 número aleatorio que representa el turno del jugador negro. Generar todo este conjunto de números aleatorios solamente se necesita hacer en el inicio del programa.

2. Para generar la clave *hash* de una posición, se comienza con una clave con valor igual a 0, y se procede a aplicar la operación *o exclusiva* (o *Xor*, $\oplus$) con los números aleatorios que se encuentren en el `ArregloZobrist` para cada pieza del tablero. Inicialmente entonces habrá que aplicar la operación $\oplus$ 32 veces, una por cada pieza. Si existe un peón al paso, también se aplica la operación con el número aleatorio generado para dicho peón al paso. Igualmente, se aplica la operación $\oplus$ para los números aleatorios que representan los derechos de enroque para cada jugador y, si el turno corresponde al jugador negro, con el número generado para el turno.

El número resultante de estas operaciones será la clave Zobrist. Dicha clave será típicamente mucho más grande que el tamaño de la tabla de transposición, por lo que solamente se utilizará el resultado de dividir con el operador módulo entre el tamaño de la tabla de transposición.

$$\texttt{indice} \leftarrow \texttt{mod(ClaveZobrist, TamañoTablaTransp).}$$

Una ventaja de este método, es que la probabilidad que dos posiciones muy similares se almacenen en una misma posición de la tabla es muy baja. Otra de sus ventajas principales es que es relativamente rápido calcular la clave Zobrist cada vez que se realiza un movimiento. Esto se debe a la forma en que la operación $\oplus$ funciona. Cada vez que un movimiento es realizado, la nueva clave puede ser calculada de la siguiente manera:

1. Se aplica la operación $\oplus$ de la posición original de la pieza que se moverá al valor actual de `ClaveZobrist`.
2. Se aplica la operación $\oplus$ con la nueva posición de la pieza con la clave Zobrist, esto añadirá de nuevo la pieza pero en una nueva posición.

Por ejemplo, si el jugador blanco tiene un peón en la posición e2 y quiere moverlo a e4, el índice de la nueva posición se calcula de la siguiente manera:

$$\begin{aligned}
&\texttt{ClaveZobrist} \leftarrow \texttt{ClaveZobrist} \oplus \texttt{ArregloZobrist[Blanco][Peón][e2]},\\
&\texttt{ClaveZobrist} \leftarrow \texttt{ClaveZobrist} \oplus \texttt{ArregloZobrist[Blanco][Peón][e4]},\\
&\texttt{indice} \leftarrow \texttt{mod(ClaveZobrist, TamañoTablaTransp).}
\end{aligned}$$

4.6.2. Información en las tablas de transposición

La información que se necesita guardar en cada entrada de la tabla hash se muestra y explica a continuación [29][22]:

Valor. El valor que se ha determinado mediante la función de evaluación sobre el nodo.

Tipo de Valor. Indica si el valor que guardamos en la posición es un valor *Minimax*, o si se trata de un límite. Si `Tabla[indice].valor`$> \beta$, entonces β será un límite inferior para el valor real. Esto indica que el movimiento ocasionó un corte durante una búsqueda anterior, pero no se sabe su valor verdadero ni es necesario determinarlo. Si `Tabla[indice].valor`$< \alpha$, entonces α será un límite superior para el valor real, y lo

único que se concluye es que el movimiento no fue lo *suficientemente bueno* para poder aumentar α.

Esto es importante ya que si pretendemos usar esta información en el futuro, es necesario poder recordar qué habíamos determinado con anterioridad para esa posición, ya que podríamos llegar a una misma posición pero con valores distintos para los límites α y β. `Tabla[indice].tipo_valor` nos indica cómo podemos hacer uso de esta información. Si `tipo_valor` es «exacto», podemos usarlo directamente, por que indica que se exploraron todos los posibles movimientos, o que recibimos una evaluación de la búsqueda *quiesence* que estuvo dentro de los límites de búsqueda α–β.

Si `tipo_valor` es «límite superior», se sabe que el valor que buscamos debe de ser menor o igual a este límite, por lo que, si el valor que se tiene guardado es menor que el valor de α en ese momento, se puede regresar este valor, generando un corte. Lo mismo pasa con un valor «límite inferior». Esto es de gran utilidad para ajustar los valores de la ventana de búsqueda.

Profundidad. Aunque encontremos una posición en la tabla, necesitamos revisar si esta información puede ser de utilidad, ya que podría ser que el valor almacenado hubiera sido calculado con una profundidad de búsqueda mas pequeña. En este caso, no podríamos usar la información. Por ejemplo, si encontramos una transposición con profundidad 5 asociado a ella, se debe a que la evaluación está basada en 5 jugadas más de búsqueda. Esto es importante, ya que si llevamos a cabo una búsqueda y encontramos esta misma posición pero a una profundidad de 8, no podremos hacer uso de la información que guardamos, ya que estos valores están basados en una profundidad de búsqueda menor y por lo tanto menos precisa.

Clave Zobrist. Debido a que dos posiciones pueden ser llegar a tener la misma posición con claves distintas (esto se explicará más adelante), se tiene que comprobar que la información guardada corresponde a la posición que nos interesa. Si la clave Zobrist de la posición actual en la búsqueda es la misma a la almacenada en la tabla de transposición, entonces se puede usar la información. De no ser así, se dice que ocurrió una *colisión*, y no podremos hacer uso de la información almacenada.

Mejor Movimiento. La tablas de transposición sirven también como un mecanismo de ordenación de los movimientos, ya que al usar profundidad iterativa, podemos identificar la secuencia de movimientos que conforman la variación principal. Esta heurística de ordenamiento (para el primer movimiento), debe de ser seguida en la próxima iteración de una búsqueda con profundidad iterativa, ya que son generalmente buenos. Aun así,

no siempre tendemos un mejor movimiento que guardar para una posición; en este caso el campo es dejado vacío sin mayor problema.

Otra ventaja de guardar estos movimientos es que podemos explorarlos antes de siquiera generar los movimientos que corresponden a esta posición, esto permite un ahorro en el tiempo de cómputo, debido a que gran parte del tiempo total de búsqueda se invierte en generar movimientos para cada posición, por lo que si tratamos este movimientos antes que todos y ocasiona un corte, no tendremos que desperdiciar tiempo generando movimientos para esa posición. Es importante mencionar que no siempre haremos uso de estos movimientos, ya que sólo serán útiles cuando no podamos hacer uso de la información que guardamos para una posición. debido a que su profunda es menor a la posición actual.

4.6.3. Uso de las tablas de tranposición

Cada vez que un movimiento es investigado en la búsqueda, se comprueba si la posición resultante existe en la tabla de transposición. Si la posición se encuentra, y se cumple que $d \leq$ `Tabla[indice].profundidad`, la información es considera útil y puede ser utilizada para producir un corte o para ajustar los límites de la ventana de búsqueda. Si la profundidad es menor, entonces se podrá hacer uso de `Tabla[indice].mejor_movimiento`, usando la tabla de transposición como el mecanismo de ordenamiento de movimientos más importante.

Cada vez que una posición es investigada durante la búsqueda α–β a cierta profundidad, ésta, es guardada en la tabla de transposición, junto con el mejor movimiento (aquel que ocasionó el corte o con el valor más alto), el valor de la posición, la profundidad de búsqueda, y el tipo de valor almacenado, indicando si se trata de un valor exacto, un límite inferior o superior. Durante la Búsqueda *Quiescence*, una posición jamás es guardada en la tabla de transposición, ya que no hay una profundidad asociada a cada posición, y no es posible saber por cuántos movimientos se extenderá dicha búsqueda.

El implementar una tabla de transposición acarrea dos tipos de problemas, el primer tipo de problemas llamado tipo 1 e identificado por Zobrist [45], es el más importante. Debido a que el número de claves hash es mucho menor que el número de posibles posiciones en el ajedrez, puede suceder que dos posiciones completamente diferentes obtengan exactamente la misma clave Zobrist. Este es un problema muy serio, porque cuando ocurre este problema, la información en la entrada puede ser usada en la posición incorrecta, provocando errores en la búsqueda. Comúnmente es posible detectar este error probando el mejor movimiento guardado

en la tabla de transposición para ver si es un movimiento legal en la posición. Si el movimiento es ilegal, entonces la entrada en la tabla debe pertenecer a otro posición diferente a la que estamos investigando.

La probabilidad de que ocurra este error depende en gran medida del tamaño de la clave *hash*, de la cual se recomienda un tamaño mínimo de 32 bits. Entre más cantidad de bits destinemos para la clave, la probabilidad de error disminuye, aunque asignar tanta memoria para las claves puede representar otro problema. El error también puede ser reducido si se cuenta con un buen generador de números aleatorios. Entre mejor sea la calidad de los números aleatorios disminuye la probabilidad de que dos posiciones distintas obtengan la misma clave.

El segundo tipo de error, llamado tipo 2, [45] , ocurre cuando diferentes claves Zobrist llevan al mismo índice para la tabla de transposición. Esto es conocido como una *colisión* [25]. Por ejemplo: si tenemos las claves *hash* 1863548654542656612 y 35498745135498755612, y un tamaño de la tabla de transposición de 1000 entradas, el índice para cada una de las claves Zobrist sería el mismo, 612.

Cuando una de estas colisiones ocurre, es necesario decidir cuál posición conservar en la tabla de transposición. A esta acción se le conoce como *políticas de reemplazo*.

Lo ideal sería poder conservar todas aquellas entradas que han resultado beneficiosas o podrían ser de utilidad en futuras búsquedas. El problema es decidir cuáles de ellas lo son y merecen la pena conservarlas, por ello se consideran diferentes políticas de remplazo:

Profundidad. La posición con la mayor profundidad es conservada [29] [22]. El concepto detrás de esta política es que la posición con la mayor profundidad usualmente tendrá más nodos que la posición con una profundidad menor. Por lo que guardar esta posición en la tabla de transposición potencialmente ahorrará mayor tiempo de cómputo que guardar aquellas posiciones investigadas a menor profundidad.

Siempre. Cada vez que ocurra una colisión, siempre se remplaza lo ya existente en la tabla con la nueva posición, sin importarnos la información que se encontrada en la entrada. Esta política se basa en que la mayoría de las transposiciones ocurren localmente en pequeños subárboles del árbol global de búsqueda [12]. Su funcionamiento es aceptable pero se sobrescribe sin importar si la información es valiosa.

Nunca. Cuando una colisión ocurra, jamás se remplazará una entrada existente en la tabla de transposición.

Profundidad – Siempre. Para implementar esta política, se utilizan 2 espacios para guardan 2 posiciones por entrada en la tabla de transposición [12]. Cuando una colisión ocurre se procede de la siguiente manera:

- Si la nueva posición ha sido explorada a profundidad mayor o igual a la almacenada en el primer espacio de la tabla de transposición, la nueva posición es guardada en este espacio y la posición vieja es movida a la segunda posición remplazando la información que se encontraba en ella.
- Si la posición es menor, esta información es guardada en la segunda posición, remplazando la información existente sin importarnos la información que contenía.

Como se puede suponer, entre mayor sea el tamaño de las tablas de transposición mayor será la cantidad de información que podremos almacenar en ellas. Esto en principio es cierto, pero por lo observado en *BuchoChess* y por [12], a partir de cierto tamaño de la tabla de transposición, un aumento considerable del tamaño de la tabla impacta poco sobre el desempeño de la búsqueda. Es importante determinar el tamaño para las tablas de transposición, de manera que se asegure un desempeño decente sin sacrificar demasiada memoria. Estudios realizados mostraron que incrementar el tamaño de una tabla al doble solamente reduce alrededor de un 5 – 7 %. Para el ajedrez, un tamaño eficiente y común para las tablas de transposición, relativamente independiente de la profundidad de búsqueda que se realizara, es alrededor de 1024k entradas [7].

Las tablas de transposición son particularmente ventajosas para métodos de búsqueda por ventana mínima junto al método de profundidad iterativa, ya que es posible llegar a explorar menos nodos que el árbol de expansión mínima. Para aprovechar las ventajas que ofrecen las tablas, éstas, tienen que ser insertadas en el algoritmo de búsqueda con ventana mínima de manera cuidadosa, la forma de hacerlo se muestra en el Algoritmo 8, el cual utiliza el Algoritmo 9 para implementar la consulta, actualización y políticas de reemplazo en las tablas de transposición.

4.7. Comentarios finales

En este capítulo se mostraron las mejoras a los algoritmos básicos que ayudan a obtener un mayor desempeño, lo que se traduce en una cantidad de nodos explorados mucho menor. Se explicaron las heurísticas usadas en el motor *BuhoChess* para obtener un mejor ordenamiento de los movimientos, debido a que el rendimiento del algoritmo de búsqueda α–β depende

fuertemente del orden en que los los nodos son explorados. También se mostraron las mejoras al algoritmo de búsqueda, como la búsqueda de aspiración y la búsqueda por ventanas mínimas, que ayudan a reducir aún más el espacio de búsqueda. Las tablas de transposición son un componente fundamental en todo juego de ajedrez ya que tiene diversas funciones, entre otras, ayuda a evitar redundancias en la búsqueda y ayudar a mejorar la ordenación de los movimientos.

Algoritmo 8 BúsquedaPVTT(α, β, d)

Entrada: $d \in \mathbb{N}$ profundidad de búsqueda

Entrada: $\alpha, \beta \in$ [`-inf`, `inf`] ▷ **Variable global** $n \leftarrow$ posición del tablero

Salida: Valor minimax de la posición actual de n ▷ Iniciar con `BúsquedaPV( -inf, inf,`d`)`

1: $ZobristKey \leftarrow$ Generar Clave Zobrist de n
2: ComprobarTT(d,α,β, $ZobristKey$)
3: **si** n es terminal **o** $d = 0$ **entonces**
4: **regresar** $-$`Quiesce(` $-\beta, -\alpha$ `)`
5: **fin si**
6: $bSearchPv \leftarrow$ **True**
7: $flag \leftarrow$ LimiteSuperior
8: $Movimientos \leftarrow$ Generar movimientos de n
9: Ordenar $Movimientos$ de acuerdo a una estrategia establecida
10: **para todo** $Movimiento \in Movimientos$ **hacer**
11: $n \leftarrow$ Hacer $Movimiento$
12: **si** bSearchPv **entonces**
13: $valor \leftarrow -$`BúsquedaPVTT(` $-\beta, -\alpha, d-1$ `)`
14: **si no**
15: $valor \leftarrow -$`BúsquedaPVTT(` $-\alpha-1, -\alpha, d-1$ `)`
16: **si** $valor > \alpha$ **entonces**
17: $valor \leftarrow -$`BúsquedaPVTT(` $-\beta, -\alpha, d-1$ `)`
18: **fin si**
19: **fin si**
20: $n \leftarrow$ Deshacer $Movimiento$
21: **si** $valor \geq \beta$ **entonces**
22: Guardar(d,α,β,valor, $ZobristKey$,$LimiteInferior$)
23: **regresar** β
24: **fin si**
25: **si** $valor > \alpha$ **entonces**
26: $\alpha \leftarrow valor$
27: $flag \leftarrow ValorExacto$
28: $bSearchPv \leftarrow$ **False**
29: **fin si**
30: **fin para**
31: Guardar(d,α,β,valor, $ZobristKey$,$flag$)
32: **regresar** α

Algoritmo 9 ComprobarTT(d,α,β, $ZobristKey$)

Salida: Valor para la posición actual

```
1: Indice ← ZobristKey % (Tamaño de la tabla Hash)
2: si TablaHash[Indice].ZobristKey == ZobristKey entonces
3:    Movimiento = TablaHash[Indice].Movimiento
4:    si TablaHash[Indice].Profundidad ≥ d entonces
5:       si TablaHash[Indice].Bandera == ValorExacto entonces
6:          regresar TablaHash[Indice].valor
7:       fin si
8:       si TablaHash[Indice].Bandera == LimiteInferior entonces
9:          si TablaHash[Indice].valor ≤ α entonces
10:            regresar α.
11:         fin si
12:         si TablaHash[Indice].valor < β entonces
13:            β ← TablaHash[Indice].valor
14:         fin si
15:      fin si
16:      si TablaHash[Indice].Bandera == LimiteSuperior entonces
17:         si TablaHash[Indice].Bandera == LimiteSuperior && TablaHash[Indice].valor ≥ β entonces
18:            regresar β.
19:         fin si
20:         si TablaHash[Indice].Bandera == LimiteSuperior && TablaHash[Indice].valor  α entonces
21:            α ← TablaHash[Indice].valor
22:         fin si
23:      fin si
24:   fin si
25: fin si
```

Capítulo 5

Poda hacia adelante

5.1. Introducción

Aún con las mejoras al algoritmo de búsqueda α–β presentadas en el capítulo anterior, existe un límite a la cantidad de cortes que es posible realizar con los métodos anteriores, y el árbol de búsqueda no puede ser reducido más de un cierto tamaño. Si bien el número de nodos explorados crece exponencialmente al incrementar la profundidad de la búsqueda, la idea es poder alcanzar mayores profundidades ya que la calidad de la decisión de los movimientos elegidos mejora al incrementar ésta [43][19].

Para reducir el número de nodos explorados, los programas de juego hacen uso de la poda hacia adelante ([29] [5]), donde un nodo es descartado, sin explorar más allá de éste, si se cree que el nodo no podrá afectar el valor *Minimax* final. Esto permite reducir aún más el árbol de búsqueda y poder llegar a mayores profundidades. Sin embargo, con este tipo de técnicas, la decisión de realizar un corte se basa en suposiciones (heurísticas) y podrían podarse ramas del árbol con buenas secuencias de movimientos, introduciendo errores en la búsqueda. Las decisiones de corte de estas técnicas se basan en los límites de búsqueda $\alpha - \beta$. Los que están enfocados en α buscan podar los nodos que no tengan el potencial en incrementar el valor de este límite, en cambio, los que están basado en β buscan podan a los nodos que excedan este límite.

Entre las técnicas de poda más populares está la *poda de movimiento nulo* que ha demostrado muy buenos resultados en el ajedrez; sin embargo, no es la única técnica de poda hacia adelante. Existen otras como *Procut* [8] y *Rankcut* [6], muy populares y efectivas en otros

juegos, aunque no tan buenas en el ajedrez. El problema de implementar las técnicas de poda hacia adelante es que normalmente éstas interfieren las unas a las otras. Por ejemplo, la *poda de movimiento nulo* y *Procut* no darán normalmente buenos resultados si se implementan las dos simultáneamente, debido a que sus criterios de corte son similares y una podará los nodos que hubiese podado la otra, limitando así su efectividad. Es importante implementar las técnica de poda que mejor resultado ofrezcan al motor de ajedrez, las cuales pueden variar de un programa a otro. En este Capítulo son estudiadas 3 técnicas de poda hacia adelante distintas, la *poda de movimiento nulo*, la *poda de inutilidad* y la *reducción de movimiento tardío*.

5.2. Poda de movimiento nulo

Esta heurística ([2], [11], [16]) basa sus decisiones de corte en un criterio dinámico, por lo que otorga al motor de ajedrez una mayor fuerza táctica en comparación con los métodos de poda estáticos. La poda de movimiento nulo está basada en la siguiente idea: en cualquier posición de ajedrez que nos encontremos, no hacer nada (hacer un movimiento nulo) sería la peor decisión posible, aunque éste fuera un movimiento legal.

Al hacer un movimiento nulo se le otorga al contrincante un movimiento más de ventaja (por lo que puede realizar dos movimientos seguidos), si no puede mejorar su posición aún con esta gran ventaja, se puede suponer que la posición era demasiado mala para él y buena para el otro jugador. Dado que ambos jugadores son óptimos, los jugadores *siempre* tratará de evitar llegar a una posición tan desventajosa para ellos, por lo que podemos ignorar esta posición y todo el subárbol que se desprende de esta posición, ya que seguramente jamás tendremos que explorarlas.

El uso de movimiento nulo solamente toma 3 % del esfuerzo empleado en toda la búsqueda y en caso que el movimiento nulo logre producir un corte podemos ahorrar un 97 % del tiempo que se hubiera requerido para explorar todo el sub-árbol [2]. En caso de que no se produzca un corte sólo se tendrá que invertir un 3 % adicional de esfuerzo computacional, lo que es un compromiso aceptable.

5.2.1. Implementación del movimiento nulo

Para implementar la poda de movimiento nulo, primero se procede a realizar un movimiento nulo, que no es otra cosa que simplemente cambiar el turno del jugador que está por mover. Debe tenerse en cuenta que no se puede hacer esto si el jugador en turno se encuentre en

jaque, ya que en el siguiente movimiento el contrincante podría capturar al rey enemigo lo que resultaría en una posición ilegal. También dos movimientos nulos seguidos están prohibidos ya que carecería de sentido [11].

Después de realizar el movimiento nulo se realiza una búsqueda a una profundidad reducida, esto se hace para tener una estimación del valor de la posición. El valor devuelto puede ser tratado como un límite inferior para la posición actual, debido a que el valor de cualquier movimiento tiene que ser mejor que el obtenido mediante el movimiento nulo. Si este valor es mayor o igual que el valor del límite superior β actual, el resultado será un corte. Si el valor es mayor que el límite inferior actual α, podemos ajustar la ventana de búsqueda, ya que el valor devuelto se convierte en el nuevo α. Si el valor es menor que el límite inferior α actual, este no contribuye en ninguna forma y habremos desperdiciado un poco de tiempo realizando el movimiento nulo.

El principal beneficio del movimiento nulo se debe a los corte que se logra, que resultan del valor devuelto de la *búsqueda reducida de movimiento nulo*. La mejor manera de conducir esta búsqueda es através de una ventana mínima alrededor del límite superior actual β, ya que solamente se busca un valor que supere al límite, y la manera más económica y rápida de hacerlo es con una búsqueda de ventana mínima. Por esta razón es que se invierte un esfuerzo de cálculo limitado para determinar si ocurrirá un corte. Una implementación típica del movimiento nulo se muestra en el Algoritmo 10.

Algoritmo 10 Poda de movimiento nulo `BúsquedaPMN(`α,β,d`)`

Entrada: $d \in \mathbb{N}$ profundidad de búsqueda
Entrada: $\alpha, \beta \in$ [`-inf`, `inf`] ▷ **Variable global** $n \leftarrow$ posición del tablero
Salida: Valor Minimax de la posición actual de n ▷ Iniciar con `BúsquedaPMN( -inf, inf,`d`)`

```
1: si n es terminal o d = 0 entonces
2:     regresar Evaluación de n
3: fin si  ▷ Conducimos una búsqueda de movimiento nulo si es legal y deseado
4: si no EnJaque() y NullOk() entonces
5:     Hacer movimiento nulo.  ▷ Se realiza una búsqueda con una ventana mínima alrededor de β
6:     valor ← −BúsquedaPMN( −β, −β + 1, d − R − 1 )
7:     Deshacer movimiento nulo  ▷ si valor > β la posición es demasiado buena, se produce un corte
8:     si valor ≥ β entonces
9:         regresar β
10:    fin si
11: fin si  ▷ Se continua con la búsqueda PVS normalmente
```

5.2.2. Problemas con el movimiento nulo

Si bien la idea en que está basada esta heurística es que no existe peor movimiento que el movimiento nulo, hay ciertas posiciones en el ajedrez donde hacer un movimiento resultaría en un deterioro la posición, ya que el jugador en turno sólo tiene malos movimientos como sus opciones legales por hacer. En estos casos, no hacer un movimiento sería la mejor opción posible. La heurística de movimiento nulo producirá cortes en estas posiciones donde una búsqueda completa no hubiera encontrado ninguna. Esto ocurre debido a que la heurística del movimiento nulo ocasiona que el programa asuma que la posición era demasiado buena cuando en realidad podría no serlo.

Un ejemplo de estas posiciones se muestra en la Figura 5.1. En esta posición la mejor opción para el jugador blanco sería no hacer nada y obtener tablas por ahogamiento ya que existe el peligro de que el peón contrario corone. Al realizar un movimiento, el jugador blanco permitiría que el peón enemigo corone lo que significaría una derrota inminente. Si durante la búsqueda se realizara un movimiento nulo el programa asumiría que la posición es demasiado buena debido a que no hacer nada es la mejor opción posible y que el valor devuelto por la heurística sería mayor que β, lo que ocasionaría un corte y provocaría que esta posición fuera descartada. De esta forma se podría perder la oportunidad de obtener tablas que es mucho mejor opción que perder la partida.

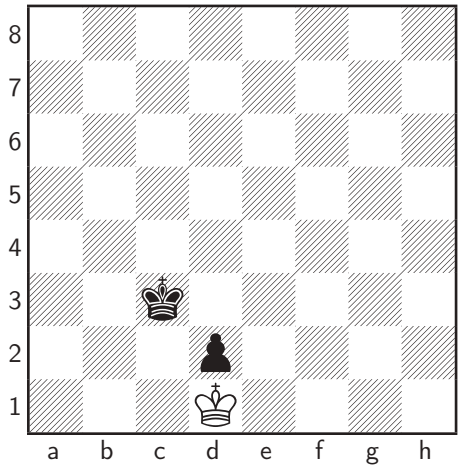

Figura 5.1: Si el jugador blanco mueve perdería la partida.

Estas posiciones son llamadas *Zugzwangan*, que significa «compulsión por moverse». Aunque las posiciones *Zugzwangan* son raras en el medio juego, son comunes en los finales de

juego, especialmente en finales donde cada uno o ambos jugadores solo cuentan con el rey y unos cuantos peones o con muy poco material. Es por ello que el movimiento nulo es evitado en posiciones de finales de juego. Para evitar usar el movimiento nulo en posiciones *Zugzwangan*, la mayoría de los programas de ajedrez ponen restricciones en su uso, estas restricciones incluyen no usar el movimiento nulo si:

- El lado a mover se encuentra en jaque.
- El lado a mover solo tiene el rey y peones.
- El lado a mover solo tiene una pequeña cantidad de piezas restantes.
- El movimiento anterior en la búsqueda también fue un movimiento nulo.

5.2.3. Factor de reducción

Como se habrá notado previamente, uno de los beneficios de la poda del movimiento nulo viene de la reducción que se realiza en la ventana mínima alrededor de β. Sin embargo, las búsquedas de profundidad reducida son responsables de debilidades debido al efecto horizonte. Estas búsquedas reducidas pueden perder información valiosa, que no serían pasadas por alto si se realizara una búsqueda completa. La búsqueda de ventana mínima se realiza a una profundidad de $d - R - 1$ donde R se conoce como el *factor de reducción*.

El tiempo que se ahorra debido a la poda de movimiento nulo depende en gran medida del factor de reducción. Una búsqueda superficial (con R grande) resulta en búsquedas más cortas y por lo tanto en árboles más pequeños de búsqueda, pero se corre el riesgo de perder información valiosa. Es por ello que en las primeras implementaciones del movimiento nulo, la mayoría de los programas usaban un factor $R = 1$ que aseguraban el menor riesgo. Con el tiempo se experimentó con valores para el factor de reducción como $R = 2$ y $R = 3$, y el que mejor compromiso ofrecía entre riesgo y ahorro de tiempo fue $R = 2$ [19].

5.3. Poda de inutilidad

La poda de inutilidad [20] está basada en podar todos aquellos movimientos que parezcan inútiles, es decir, que no tengan potencial de incrementar el valor de α. Al basar sus decisiones de poda alrededor del límite inferior α, esta heurística puede implementarse sin problemas junto con otros heurísticas como *Movimiento nulo* que basa sus decisiones de corte alrededor

de β. Estás heurísticas no interfirirán entre ellas ya que tratan de podar diferentes tipos de nodos.

Existen varias versiones de esta poda, en su forma más pura, se aplica en los nodos frontera, aquellos con profundidad 1. Si se aplica en los nodos pre-frontera a profundidad 2, se le conoce como *Razoning* [20]. Para hacer la poda, la heurística toma en cuenta que la función de evaluación puede ser separada en dos tipos de componentes: las de mayor importancia y las complementarias. Por ejemplo, una de las componentes más importantes al momento de evaluar una posición es el balance material, mientras que entre las complementarias se incluyen factores posicionales.

La idea es realizar un movimiento y evaluar la nueva posición. Al evaluar la posición solamente se toma en cuenta las componentes más importantes de la evaluación, ya que estas son las que provocan normalmente los grandes cambios en la función. Si la evaluación parcial es muy baja y queda fuera de los límites de búsqueda, entonces se considera un margen de error o margen de utilidad. Este margen representa el valor posicional más grande que se podría obtener.

Si aún con el margen de utilidad no puede superar el límite inferior α es posible considerar que no es un movimiento lo suficientemente bueno, y que no lo será aunque se realice la búsqueda en el subárbol, por lo que es candidato a un corte. El algoritmo de la poda de inutilidad se muestra en el Algoritmo 11.

Existen ciertas restricciones de uso de la heurística. No es usada cuando el lado a mover se encuentra en jaque, o cuando alguno de los límites α o β se encuentran cerca del valor de $Mate$ o $-Mate$, debido a que podría dejar al programa ciego ante ciertos *jaque mates*. Tampoco debe usarse si el movimiento realizado provoca un jaque al rey enemigo ya se trata de un movimiento que es necesario revisarlo más cuidadosamente. Todas esas posiciones se consideran demasiado dinámicas como para poder descartarlas, por lo que si se aplicara ocasionaría errores en la búsqueda.

Los valores del margen de utilidad varían para cada programa, ya que el valor del margen está relacionado a la función de evaluación. Normalmente se utilizan valores de 2-5 peones [20]. Si se aplica en los nodos frontera, el valor de 2 peones normalmente es suficiente, si se aplica en nodos pre–frontera se necesitan valores más grandes, alrededor del valor de una torre o 5 peones. El margen necesita ser más grande en ese caso debido a que estamos podando 2 profundidades de búsqueda y el cambio en la función de evaluación puede variar más. Podas a esta profundidad, sin embargo, suponen un riesgo más grande ya que podemos perder infor-

mación importante. Para la poda en nodos frontera se ha mostrado que esta heurística resulta ser bastante segura en la práctica [20].

5.4. Reducción de movimiento tardío

Esta heurística basa su decisión de corte en una buena ordenación de los movimientos y poda aquellos movimientos que normalmente no tendrán posibilidades en aumentar el valor de α. Los movimientos que son considerados buenos se ordenan siempre al principio en la lista de movimientos y normalmente producirán buenos resultados y cortes en la búsqueda. Por otro lado, un movimiento tardío es un movimiento que se ordena hasta el final de la lista de movimientos que estamos apunto de explorar.

Estos movimientos raramente producen buenos resultados. Por ejemplo, en algunas ordenaciones de movimiento se consideran a los movimientos que pierden material para ser explorados al final. Estos movimientos necesitan condiciones muy específicas en la posición para que produzcan buenos resultados, tales como, sacrificar una pieza para exponer el Rey enemigo o atacar a una pieza desprotegida.

La idea básica de la reducción del movimiento tardío, es la predicción del valor este tipo de movimientos. Como normalmente no producen buenos resultados, no tendrán potencial de incrementar α y se podría reducir el tiempo que se ocupa en explorar esos nodos. Esto es opuesto a la *poda de movimiento nulo* que trata de encontrar valores que superen el límite de β. Debido a que estas técnicas operan en diferentes tipos de nodos, no se traslaparán, por lo que pueden implementarse en conjunto sin problema. Se muestra el seudocódigo de la reducción de movimiento tardío en el Algoritmo 12.

La reducción del movimiento tardío realiza una búsqueda de ventana mínima por cada movimiento poco prometedor, debido a que sólo se desea comprobar si el movimiento puede superar el límite inferior α. Para realizar esta búsqueda se hace uso de una reducción en la profundidad de búsqueda. Si la evaluación regresa una valor inferior a α indica que el movimiento fue pobre y no valdrá la pena invertir más tiempo determinando su valor. Si la búsqueda regresa un valor superior a α, indica que existe algo interesante en la posición, y se tiene que realizar la búsqueda con los límites reales de la ventana de búsqueda y utilizar la profundidad regular.

La idea principal de esta heurística es comenzar explorando todos los movimientos prometedores a una profundidad complementa. Entre estos movimientos se consideran los movimientos

hash, capturas que ganen material, capturas que conserven material y los movimientos asesinos. Todos los demás movimientos son explorados con una profundidad reducida. Generalmente los primeros 3 a 5 movimientos son una buena cantidad para explorar a profundidad completa antes de reducir la profundidad de búsqueda.

Existen ciertas condiciones donde no es aconsejable aplicar la reducción de movimiento tardío, debido a los riesgos que suponen. Entre los tipos de movimientos que no deberíamos considerar en reducir se encuentran:

1. Jaques. Normalmente un jaque provoca extensiones en la búsqueda, por lo que si reducimos la búsqueda, estas extensiones son canceladas y carecen de sentido.

2. Capturas. No deberían nunca reducirse capturas perdedoras de material ya que podría incrementar el riesgo del efecto horizonte.

3. Promociones. Normalmente las promociones son movimientos lo suficientemente interesantes, y vale la pena tratar de estimar el valor para estas posiciones, por lo que no deben de ser reducidas.

Estas son las condiciones básicas para evitar reducir movimientos riesgosos o interesantes. Sin embargo, existen otras técnicas para determinar si un movimientos es seguro para reducir. Existen implementaciones de esta heurística donde se conserva un valor histórico para cada movimiento, donde se toman en cuenta que tan útiles han sido dichos movimientos. Si estos han sido de mucha utilidad se evita reducir estos movimientos.

5.5. Comentarios finales

En este capítulo, se presentaron algunas de las técnicas más actuales que se utilizan en búsquedas para juegos deterministas de suma cero. Todas las técnicas presentadas en este capítulo reducen el espacio de búsqueda por debajo de la técnica α–β. En el motor de ajedrez desarrollado *Buhochess* se implementaron los métodos de poda hacia adelante presentados con el fin de evaluar las ventajas y desventajas que presentan estos métodos. En todos los casos, estos métodos incrementaron en forma significativa el poder de juego de *Buhochess*

Si bien, estas heurísticas permiten realizar la búsqueda a una mayor profundidad y por lo tanto elevar el poder de juego del motor de ajedrez, su uso implica tácitamente renunciar a la optimalidad del algoritmo α–β. Debido a que estos métodos utilizan heurísticas para eliminar

sin comprobación alguna ramas completas del árbol de juego, la calidad de la búsqueda depende no solamente de la función objetivo a maximizar (la función de evaluación) si no también a los criterios impuestos para la poda hacia adelante.

Algoritmo 11 Poda de Inutilidad `BúsquedaPI(` α`,` β`,`d`)`

Entrada: $d \in \mathbb{N}$ profundidad de búsqueda

Entrada: $\alpha, \beta \in$ [`-inf`, `inf`] ▷ **Variable global** $n \leftarrow$ posición del tablero

Salida: Valor minimax de la posición actual de n ▷ Iniciar con `BúsquedaPI( -inf, inf,`d`)`

1: **si** n es terminal **o** $d = 0$ **entonces**
2: **regresar** Evaluación de n
3: **fin si**
4: $bSearchPv \leftarrow$ **True**
5: $Poda \leftarrow$ **False**
6: $Movimientos \leftarrow$ Generar movimientos de n
7: Ordenar $Movimientos$ de acuerdo a una estrategia establecida ▷ Poda de inutilidad.
8: $Valor\_Material \leftarrow$ Cacular balance material.
9: **si no** `Jaque(`n`)` **y** $d == 1$ **y** $(Valor\_Material + 200) \leq \alpha$ **entonces**
10: $Poda \leftarrow true.$
11: **si no** y **si no** `Jaque(`n`)` **y** $d == 2$ **y** $(Valor\_Material + 500) \leq \alpha$ **entonces**
12: $Poda \leftarrow true$
13: **fin si**
14: **para todo** $Movimiento \in Movimientos$ **hacer**
15: $n \leftarrow$ Hacer $Movimiento$
16: **si** $Poda$ **y no** `Jaque(`n`)` **entonces**
17: $n \leftarrow$ Deshacer $Movimiento$
18: $Continuar$
19: **fin si**
20: **si** bSearchPv **entonces**
21: $valor \leftarrow -$`BúsquedaPI(` $-\beta, -\alpha, d-1$ `)`
22: **si no**
23: $valor \leftarrow -$`BúsquedaPI(` $-\alpha-1, -\alpha, d-1$ `)`
24: **si** $valor > \alpha$ **entonces**
25: $valor \leftarrow -$`BúsquedaPI(` $-\beta, -\alpha, d-1$ `)`
26: **fin si**
27: **fin si**
28: $n \leftarrow$ Deshacer $Movimiento$
29: **si** $valor \geq \beta$ **entonces**
30: **regresar** β.
31: **si no** y **si** $valor > \alpha$ **entonces**
32: $\alpha \leftarrow valor$
33: $bSearchPv \leftarrow$ **False**
34: **fin si**
35: **fin para**
36: **regresar** α.

Algoritmo 12 Reducción de movimiento tardio `BúsquedaRMT(`α`,`β`,`d`)`

Entrada: $d \in \mathbb{N}$ profundidad de búsqueda

Entrada: $\alpha, \beta \in$ [`-inf`, `inf`] ▷ **Variable global** $n \leftarrow$ posición del tablero

Salida: Valor minimax de la posición actual de n ▷ Iniciar con `BúsquedaRMT( -inf, inf,`d`)`

si n es terminal **o** $d = 0$ **entonces**
  regresar Evaluación de n.
fin si
$bSearchPv \leftarrow$ **True**
$Movimientos \leftarrow$ Generar movimientos de n
Ordenar $Movimientos$ de acuerdo a una estrategia establecida
para todo $Movimiento \in Movimientos$ **hacer**
  $n \leftarrow$ Hacer $Movimiento$
  si bSearchPv **entonces**
    $valor \leftarrow -$`BúsquedaPV(`$-\beta, -\alpha, d-1$`)`
  si no
    si $Movientos\_Explorados \geq 4$ **y** $d \geq 3$ **y** `Reducir_Ok(`n`)` **entonces**
      $valor \leftarrow -$`BúsquedaRMT(`$-\alpha, -\alpha+1, d-2$`)`
    si no
      $\alpha \leftarrow \alpha + 1$
    fin si
    si $valor > \alpha$ **entonces**
      $valor \leftarrow -$`BúsquedaRMT(`$-\alpha-1, -\alpha, d-1$`)`
      si $valor > \alpha$ **entonces**
        $valor \leftarrow -$`BúsquedaRMT(`$-\beta, -\alpha, d-1$`)`
      fin si
    fin si
  fin si
  $n \leftarrow$ Deshacer $Movimiento$
  si $valor \geq \beta$ **entonces**
    regresar β.
  si no y **si** $valor > \alpha$ **entonces**
    $\alpha \leftarrow valor$
    $bSearchPv \leftarrow$ **False**
  fin si
  $Movientos\_Explorados \leftarrow Movientos\_Explorados + 1$
fin para
regresar α.

Capítulo 6

Resultados experimentales

6.1. Introducción

Con el fin de medir y obtener una idea de cómo las heurísticas que se implementaron en el motor *Buhochess* contribuían a la reducción del árbol de juego, se realizaron algunas pruebas experimentales. Numerosas búsquedas fueron llevadas a cabo a profundidades predefinidas en el conjunto de posiciones de prueba *LCT–II–Test11* [26], el cual contiene 35 posiciones claves para probar diferentes heurísticas y algoritmos. De éstas, se seleccionaron 3 posiciones de juego medio y 3 de finales para las pruebas. De esta manera se pretende verificar si la fase de juego en que se aplican las heurísticas afectaba su desempeño. No se escogieron posiciones de inicio debido a que normalmente puede ser calculadas mediante un libro de aperturas.

Para medir el desempeño de las heurísticas, se procedió a contar los nodos generados por cada una de ellas en una misma posición. Esta forma de medir el desempeño tiene una ventaja clara, y es que los resultados pueden ser medidos de forma objetiva, sin la necesidad de comprobar el poder de juego del programa *BuhoChess*. Medir el poder de juego de un problema es una tarea subjetiva y que depende de las habilidades del jugador específico.

Los experimentos realizados fueron llevadas a cabo en diferentes computadoras y plataformas, para tratar de llevarlas a cabo simultáneamente, debido a la gran cantidad de tiempo que necesitaba cada uno de ellas. Se usaron 5 computadoras con las mismas características. El motor *BuhoChess* se programó en *Visual C++* bajo el entorno *.net*, prestando especial atención al desempeño y velocidad del programa.

6.2. Comparación de heurísticas

El objetivo del primer experimento es evaluar el desempeño de las diferentes heurísticas comparadas a una versión de *BuhoChess* que solamente usa la heurística de poda $\alpha - \beta$. Una dificultad que surgió al realizar el experimento, es debido a la extrema diferencia en desempeño entre el uso de la poda α–β con el uso de diferentes heurísticas. Según los resultados teóricos obtenidos, la diferencia en cuanto a nodos generados crece en forma exponencial conforme aumenta la profundidad de la búsqueda. Es por ello que la comparación sólo fue posible a profundidades relativamente pequeñas.

El experimento consiste en comparar diferentes versiones del motor de búsqueda *BuhoChess*, las cuales se etiquetaron desde la letra A a la letra H. Estas versiones son incrementales con respecto a su predecesora, por ejemplo: la versión D, implementa no sólo las heurísticas descritas, sino también todas las heurísticas mencionadas en los incisos anteriores. Las versiones implementadas son:

A: Poda $\alpha - \beta$.

B: Heurísticas de ordenación de movimientos estáticas y dinámicas, así como la tabla transposición. Sin embargo, la tabla de transposición solamente es usada para la ordenación de movimientos. También se hace uso de la profundidad iterativa ya que es necesaria para las heurísticas dinámicas.

C: Algoritmo de ventanas mínimas.

D: Tablas de transposición para la evaluación y el uso de variación principal.

E: Búsqueda de aspiración.

F: Poda de movimiento nulo.

G: Poda de inutilidad.

H: Reducción de movimiento tardío.

Para presentar los resultados se muestra dos gráficas: la Figura 6.1 representa una posición de juego medio y la Figura 6.2 representa una posición de finales. Las posiciones mostradas se considera representativas de los resultados obtenidos. La gráfica presenta el decremento en número de nodos explorados al usar las diferentes heurísticas. Está expresada en escala logarítmica debido a las grandes diferencias entre sus valores. En la gráfica el 100 % representa

el número de nodos explorados por la poda α–β a una nivel de profundidad. El resto de los resultados se muestran en las tablas en el apéndice B.

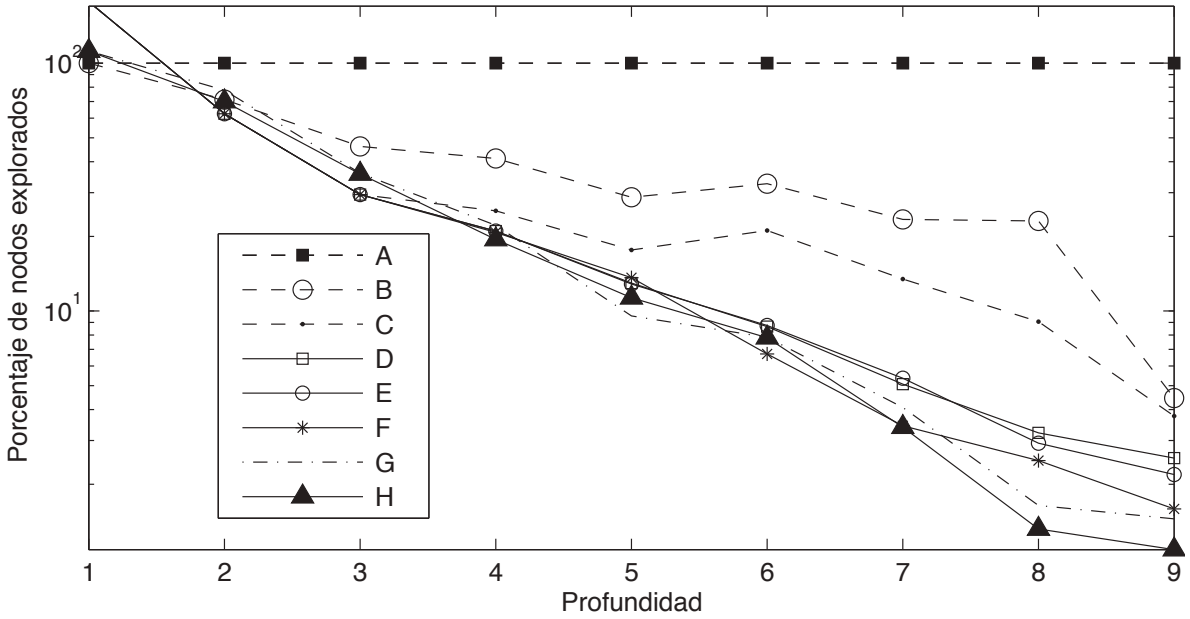

Figura 6.1: Gráfica de los resultados obtenidos de la posición 2.

Como se esperaba, la ordenación de los movimientos tiene un fuerte efecto en el desempeño del algoritmo α–β, ya que permite reducir en gran medida el tamaño del árbol de búsqueda. Para solamente una profundidad de 4 se logra reducir un 80 % el número de nodos que su hubieran explorado con α–β, puro. Para profundidades de 8 en adelante esta reducción es mayor al 90 %. Esto sugiere la fuerte dependencia en las heurísticas de ordenamiento de los algoritmos de búsqueda basados en un marco α–β.

Cualquier esfuerzo extra invertido para obtener una buena ordenación de los nodos, como ocurre con la profundidad iterativa, está por demás justificado. Al contar con una buena ordenación del árbol a explorar, la búsqueda por ventanas mínimas ofreció un desempeño favorable en todos las posiciones y fases del juego analizadas. Este ahorro se estima alrededor del 13 % en profundidades mayores a 5, lo que justifica el hecho que sea el algoritmo de búsqueda más empleado en los motores comerciales de ajedrez.

Otra heurística que ofreció buenos resultados es la búsqueda por aspiración. Su desempeño fue pobre en posiciones de juego medio si se le compara con otras heurísticas. Sin embargo, nunca implicó un mayor costo computacional, y en un motor de ajedrez cualquier ahorro en tiempo es aceptable, ya que podría significar un nivel más de búsqueda. Su bajo desempeño

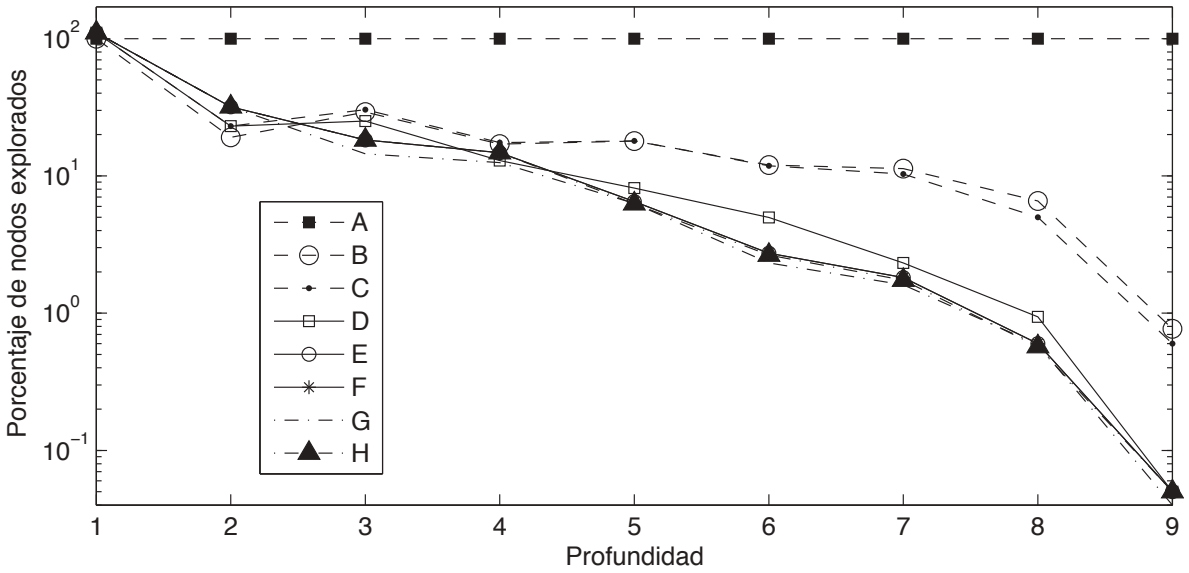

Figura 6.2: Gráfica de los resultados obtenidos de la posición 4.

se debe a la fase y posición de juego medio, ya que normalmente ocurren grandes cambios en la función de evaluación, lo que provocará muchas búsquedas extras por esta heurística. Sin embargo, para finales de juego, ésta ofreció ahorros de tiempo cercanos al 7 %. Ésto, puede deberse a que, en esta fase del juego, los cambios en la función de evaluación son menores, debido a la escases de material. Lo que provoca que se pueda estimar una mejor ventana de búsqueda para la búsqueda de aspiración.

Las tablas de transposición juegan un papel fundamental de cualquier juego de ajedrez ya que evitan la redundancia en la búsqueda y ayudan a ajustar los límites de la misma. Esto se tradujo en un 20 % a 25 % de ahorro en tiempo en las pruebas. Además se pudo observar que las tablas resultan extremadamente efectivas en los finales del juego. Las tablas aportan alrededor de un 60 % a 70 % de ahorro en esta fase.

Los resultados muestran que con el sólo uso de tablas de transposición, búsqueda de aspiración, ventanas mínimas y una buena ordenación de los nodos es posible ahorrar un 96 % del tiempo de cómputo respecto al algoritmo α–β. Al ser éste el límite máximo en el ahorro de tiempo posible con un algoritmo basado en α–β, es necesario el uso de técnicas de poda hacia adelante, si se desea realizar búsquedas a mayor profundidad, y por consecuencia elevar el nivel de juego del motor de ajedrez *Buhochess*.

La poda de movimiento nulo otorgó un ahorro cercano del 20 % al 26 % en posiciones de juego medio. Se observó que su efectividad depende en gran medida de la profundidad en que

es empleada, ya que para profundidades menor a 4 no reduce (y en algunos casos aumenta) el número de nodos explorados. Para profundidades mayores a ésta, su efectividad incrementa. Los experimentos realizados mostraron que esta heurística es poco efectiva en las finales de juego, debido que en las condiciones que necesita para realizarse no se cumplen, por lo que casi nunca se llevará acabo. Esto se puede apreciar en la Figura 6.2.

De igual manera, la poda de inutilidad demostró resultados favorables en el juego medio, produciendo un ahorro promedio del 13 %. Sin embargo aunque su uso no está prohibido en el final de juego, su efectividad se ve disminuida casi por completo en esta fase. Se concluye que se debe a que muchos de los factores importantes en la función de evaluación se ven disminuidos, por ejemplo: el balance material. Y como la heurística depende en gran medida de esos factores, muy pocos o ningún resultados favorables son obtenidos en las pruebas. Esto se aprecia en la Figura 6.2.

La reducción de movimiento tardío ahorra en promedio 6 % a 10 % del tiempo en posiciones de juego medio, lo que es un ahorro aceptable. Este se desempeña bien debido a la gran cantidad de malos movimientos o inútiles que reducir. Sin embargo, en la fase final, cuando existe poca cantidad de piezas y por ende de movimientos, ésta ve su efectividad reducida. Muchas veces la heurística no se aplicará, ya que explora los primeros n movimientos (normalmente 4 a 6) a profundidad máxima, por lo que si la cantidad de movimientos es menor a n, la heurística no hará nada. En general está heurística se comporta mal en el final de juego.

6.3. Ajustes de tablas de transposición

Debido a la importancia de las tablas de transposición en cualquier programa de ajedrez, es importante determinar ciertos criterios para obtener su máximo desempeño. Es por ello que el objetivo de este experimento es determinar el efecto de los diferentes esquemas de reemplazo de información en una tabla de transposición cuando una colisión ocurre. También pretende determinar el papel que juega el tamaño de las tablas para reducir el número de colisiones. Para este experimento, se seleccionó una posición de final de juego, mostrada en la Figura B.5, debido a que las tablas son de más utilidad en la fase final del juego, por lo que será mucho más fácil medir los resultados.

El experimento consiste en realizar una búsqueda a partir de la posición variando la profundidad de la búsqueda de 1 hasta 12 con los cuatro esquemas de remplazo mencionados en este trabajo: *siempre, nunca, profundidad, siempre–profundidad*. Para cada esquema se utilizaron 6 tamaños de tabla, comenzando en 128k y doblando el valor cada vez, con lo que los tamaños

de la tabla son: 128K, 256K, 512K, 1024K, 2048K entradas en la tabla. Las tablas con los resultados obtenidos para cada una de los diferentes esquemas de remplazo se presentan en el apéndice B.

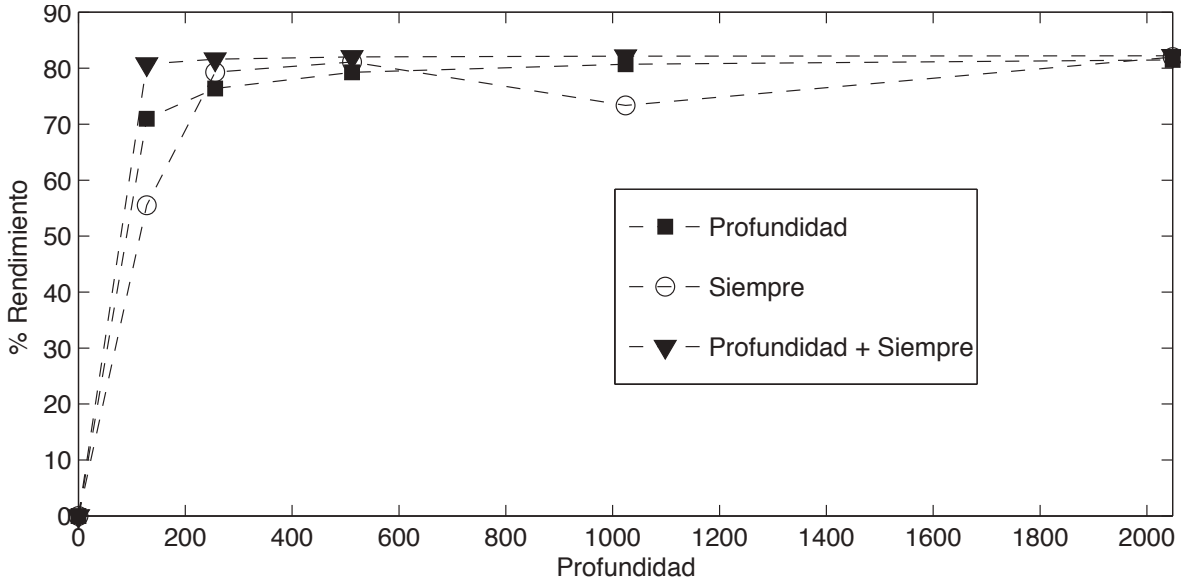

Figura 6.3: Aumento del rendimiento en relación a la memoria.

La gráfica que se muestra en la Figura 6.3 representa el aumento en el desempeño al doblar el tamaño de la tabla para cada una de los esquemas de remplazo. A medida que el tamaño de las tablas de transposición crece, se puede observar que el número de nodos explorados tiende a estabilizarse. Esto se debe a que después de un tamaño (en el caso del motor *Buhochess* después de 2048 K entradas), ningún ahorro significativo se obtiene al incrementar el tamaño de la tabla de transposición.

Esto es causado por el número máximo de nodos que pueden ser guardados en una tabla de transposición sin ocasionar colisiones significativas que afecten su desempeño. El ahorro que se logra a medida que se incrementa al doblar el tamaño de entradas en las tablas superiores de 512K es solamente alrededor 5 % al 7 % en este caso. Para tablas de menor tamaño de 128K donde las colisiones son un poco más comunes, el aumento del tamaño supuso un aumento del desempeño de 15 %. Se puede concluir que un buen tamaño para las tablas de transposición que tenga un buen desempeño en relación a la memoria es alrededor de 1024K.

De los esquemas de reemplazo, el esquema *nunca* fue el que peores resultados mostraba, ya que el aumento en el tamaño de su tabla no afectaba en lo más mínimo a la cantidad de nodos a explorar. El uso de este esquema no ahorró ningún nodo en la búsqueda.

El segundo en desempeñó fue el esquema *siempre*, con 35 % de aumento. Este desempeño es muy fluctuante y se puede apreciar en la Figura 6.3, debido que a veces puede llegar a explorar menos nodos que otros esquemas de reemplazo y a veces requiere explorar más. Se piensa que esto se debe a que se sobreescribe la información sin importarle lo útil que esta sea, se podría decir que este esquema implica un poco de azar.

El tercer esquema fue el basado en la profundidad, ahorra cera de un 67 % de los nodos. Debido a que normalmente una profundidad mayor indica un sub árbol más grande atado a esa posición, cada vez que sea encontrado un corte ahorrará mucho más tiempo. Sin embargo, este esquema termina por guardar información que casi nunca más será utilizada, debido a que se guardan siempre profundidades mayores.

El último esquema y por mucho el mejor fue *profundidad+siempre*. Esto se puede deber a que guarda el doble de información que las demás, y evitar las desventajas de los esquemas *siempre* y *profundidad*. Igualmente, este esquema combina sus ventajas como: la de guardar la información más reciente ya que es problable que vuelva ser usada en pocos movimientos más; y guardar la información más valiosa basado en el sub árbol atado a ésta. El esquema supuso serca de un 84 % de ahorro de nodos explorados.

6.4. Comentarios finales

En esta sección se presentaron algunas pruebas experimentales sobre el uso de diferentes heurísticas y su impacto en la búsqueda, aplicada al motor de ajedrez *Buhochess*. Con el fin de presentar solamente las ideas principales sólo se seleccionaron los resultados más representativos y se presentaron y discutieron en forma de gráfica. Los resultados detallados para diversas posiciones se presentan en el apéndice B. En general todas las heurísticas presentan beneficios a lo largo de la partida, aunque algunas son más notorias en los finales, como el uso de ventanas de aspiración y tablas de transposición. Por otra parte, los métodos de poda hacia adelante, se revelaron como técnicas útiles para el juego medio, que deben ser evitados en general en la fase de finales de partida.

Siendo las tablas de transposición una de las técnicas más sensibles e importantes en el desarrollo de los algoritmos de búsqueda, se realizó un estudio sobre la importancia del tamaño de la tabla de transposición, así como el uso de las diferentes estrategias de reemplazo. A partir de dichos experimentos se estableció el tamaño de la tabla de transposición a 2048 K entradas utilizando la estrategia de *siempre–profundidad*.

Es importante mencionar que los valores obtenido obtenidos para el desempeño en *Buho-Chess* no son absolutos ya que éstos pueden variar en cada implementación, posición y sobre todo por la función de evaluación. Siendo esta una evaluación muy ligada a la implementación en específico, estos resultados se presentan como ejemplos ilustrativo de las diferentes heurísticas y algoritmos desarrollados y sus resultados deben ser considerados más como una guía que como reglas generales.

Capítulo 7

Conclusiones y trabajos futuros

El principal objetivo de este trabajo era estudiar e implementar los algoritmos de búsqueda más usados en la actualidad por los programas de ajedrez, para desarrollar desde cero, un motor de ajedrez propio al que se le llamó *BuhoChess*.

El desarrollo se centró en dos partes: la primera consistió en desarrollar rutinas eficientes. Esto es importante, ya que cualquier ahorro en el tiempo puede significar el poder alcanzar un nivel más de búsqueda. Si bien el poder de cálculo del programa es de 210 nodos por segundo, la cual se logró principalmente por la implementación de la representación 0x88.

La segunda fue implementar y estudiar los algoritmos y heurísticas más usados en los programas de ajedrez en la actualidad. En un principio se implementó el algoritmo α–β así como sus variantes, que permitieron reducir la cantidad de nodos explorados hacia un número cercano al árbol de expansión mínimo. Esto es, el mínimo posible de nodos que aseguren que el movimiento seleccionado es óptimo de acuerdo al criterio *Minimax* y a la función de evaluación.

No sólo se pretendía reducir el mayor número posibles de nodos en la búsqueda utilizando el algoritmo α–β y sus modificaciones, sino que también se implementaron técnicas que permitieron reducir aún más esta cantidad, incluso menor que el árbol de expansión mínimo, a costa de perder la optimalidad del algoritmo. Con estas técnicas de poda hacia adelante, se pueden lograr profundidades de búsqueda importantes, y ofrecer un mayor poder de juego.

Uno de los los principales desafíos a los que se enfrentó fue el de tratar de optimizar los parámetros de las heurísticas dinámicas, ya que estas tienden a desempeñarse de formas muy

diferentes dependiendo de las condiciones donde se apliquen, como la posición, la profundidad de búsqueda o la función de evaluación usada entre otros. Existen multitud de estos parámetros para optimizar en todas las heurísticas, que van desde el orden en que se consideran los movimientos, tamaño de las tablas, peso de un movimiento suficiente hasta el margen de error usado. Si bien éstos se establecieron con la ayuda de la bibliografía, algunas fueron ajustadas a partir de una etapa de experimentación con el motor de ajedrez desarrollado.

Debe tenerse en cuenta el conjunto de heurísticas que se usarán en el motor, ya que no por usar las heurísticas más novedosas o las mayormente utilizadas se garantizará un buen desempeño. Es importante probar en la implementación específica que combinaciones de éstas deben usarse para obtener mejores resultados. En el caso de *BuhoChess* se implementaron y experimentaron todas las heurísticas estudiadas en este trabajo.

La optimalidad en el juego de acuerdo al criterio *Minimax* está dado en relación a la función de evaluación. Para el desarrollo de *BuhoChess* se realizó un estudio de las características más importantes para la evaluación de una posición particular y se aplicó una función de evaluación lineal en relación a las características seleccionadas. Algunos de estos criterios cambian de peso de acuerdo a la fase de juego.

El poder de juego de *BuhoChess* se logró gracias a la búsqueda bibliográfica de los valores recomendados para la función de evaluación y, sobre todo, a la experiencia en el juego del autor para ajustar estos pesos. Estos criterios son subjetivos y es necesaria una experimentación intensiva para validarlos. Por otra parte, el uso de una función de evaluación lineal es una simplificación de la manera en que un humano evalúa las posiciones de un tablero.

Como trabajo futuro entonces, es primordial el desarrollo de una función de evaluación más compleja y que tome en cuenta aspectos no lineales. Dicha función puede desarrollarse como una red neuronal, donde las neuronas de entrada son los valores (en *centipawns*) de las diferentes características, y como neurona de salida el valor de la posición. El uso de una red neuronal permite el uso de técnicas de aprendizaje supervisado y aprendizaje reforzado para ajustar la función de evaluación.

Apéndice A

Detalles de la función de evaluación

A.1. Matrices de evaluación de posición de pieza

La posición de las piezas se evalúa en relación a las matrices siguientes, desde el punto de vista del jugador blanco:

Peón:

$$\begin{bmatrix} 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0, \\ 5, & 10, & 15, & 20, & 20, & 15, & 10, & 5, \\ 4, & 8, & 12, & 16, & 16, & 12, & 8, & 4, \\ 3, & 6, & 9, & 12, & 12, & 9, & 6, & 3, \\ 2, & 4, & 6, & 8, & 8, & 6, & 4, & 2, \\ 1, & 2, & 3, & -10, & -10, & 3, & 2, & 1, \\ 0, & 0, & 0, & -40, & -40, & 0, & 0, & 0, \\ 0, & 0, & 0, & 0, & 0, & 0, & 0, & 0 \end{bmatrix}$$

Caballo:

$$\begin{bmatrix} -10 & -10 & -10 & -10 & -10 & -10 & -10 & -10 \\ -10 & 0 & 0 & 0 & 0 & 0 & 0 & -10 \\ -10 & 0 & 5 & 5 & 5 & 5 & 0 & -10 \\ -10 & 0 & 5 & 10 & 10 & 5 & 0 & -10 \\ -10 & 0 & 5 & 10 & 10 & 5 & 0 & -10 \\ -10 & 0 & 5 & 5 & 5 & 5 & 0 & -10 \\ -10 & 0 & 0 & 0 & 0 & 0 & 0 & -10 \\ -10 & -30 & -10 & -10 & -10 & -10 & -30 & -10 \end{bmatrix}$$

Alfil:

$$\begin{bmatrix} -10 & -10 & -10 & -10 & -10 & -10 & -10 & -10 \\ -10 & 0 & 0 & 0 & 0 & 0 & 0 & -10 \\ -10 & 0 & 5 & 5 & 5 & 5 & 0 & -10 \\ -10 & 0 & 5 & 15 & 15 & 5 & 0 & -10 \\ -10 & 0 & 5 & 15 & 15 & 5 & 0 & -10 \\ -10 & 0 & 5 & 5 & 5 & 5 & 0 & -10 \\ -10 & 0 & 0 & 0 & 0 & 0 & 0 & -10 \\ -10 & -10 & -20 & -10 & -10 & -20 & -10 & -10 \end{bmatrix}$$

Torre:

$$\begin{bmatrix} -6 & -3 & 0 & 3 & 3 & 0 & -3 & -6 \\ -6 & -3 & 0 & 3 & 3 & 0 & -3 & -6 \\ -6 & -3 & 0 & 3 & 3 & 0 & -3 & -6 \\ -6 & -3 & 0 & 3 & 3 & 0 & -3 & -6 \\ -6 & -3 & 0 & 3 & 3 & 0 & -3 & -6 \\ -6 & -3 & 0 & 3 & 3 & 0 & -3 & -6 \\ -6 & -3 & 0 & 3 & 3 & 0 & -3 & -6 \\ -6 & -3 & 0 & 3 & 3 & 0 & -3 & -6 \end{bmatrix}$$

Reina:

$$\begin{bmatrix} -24, & -16 & -12 & -8 & -8 & -12 & -16 & -24 \\ -16 & -8 & -4 & 0 & 0 & -4 & -8 & -16 \\ -12 & -4 & 0 & 4 & 4 & 0 & -4 & -12 \\ -8 & 0 & 4 & 8 & 8 & 4 & 0 & -8 \\ -8 & 0 & 4 & 8 & 8 & 4 & 0 & -8 \\ -12 & -4 & 0 & 4 & 4 & 0 & -4 & -12 \\ -16 & -8 & -4 & 0 & 0 & -4 & -8 & -16 \\ -24 & -16 & -12 & -8 & -8 & -12 & -16 & -24 \end{bmatrix}$$

Rey:

$$\begin{bmatrix} -40 & -40 & -40 & -40 & -40 & -40 & -40 & -40 \\ -40 & -40 & -40 & -40 & -40 & -40 & -40 & -40 \\ -40 & -40 & -40 & -40 & -40 & -40 & -40 & -40 \\ -40 & -40 & -40 & -60 & -60 & -40 & -40 & -40 \\ -40 & -40 & -40 & -60 & -60 & -40 & -40 & -40 \\ -40 & -40 & -40 & -40 & -40 & -40 & -40 & -40 \\ -20 & -20 & -20 & -20 & -20 & -20 & -20 & -20 \\ 0 & 20 & 40 & -20 & 0 & -20 & 40 & 20 \end{bmatrix}$$

Rey al final de la partida. Sustituye a la característica de protección del rey sobre la estructura de los peones del rey.

$$\begin{bmatrix} 0 & 10 & 20 & 30 & 30 & 20 & 10 & 0 \\ 10 & 20 & 30 & 40 & 40 & 30 & 20 & 10 \\ 20 & 30 & 40 & 50 & 50 & 40 & 30 & 20 \\ 30 & 40 & 50 & 60 & 60 & 50 & 40 & 30 \\ 30 & 40 & 50 & 60 & 60 & 50 & 40 & 30 \\ 20 & 30 & 40 & 50 & 50 & 40 & 30 & 20 \\ 10 & 20 & 30 & 40 & 40 & 30 & 20 & 10 \\ 0 & 10 & 20 & 30 & 30 & 20 & 10 & 0 \end{bmatrix}$$

Apéndice B

Resultados detallados de las pruebas realizadas

B.1. Posiciones utilizadas pata la evaluación

B.1.1. Posiciones de media partida

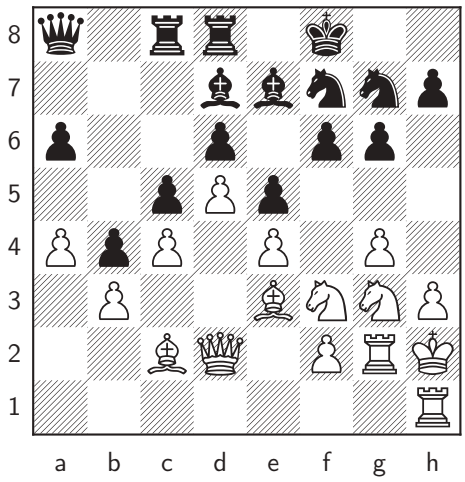

Figura B.1: Posición 1.

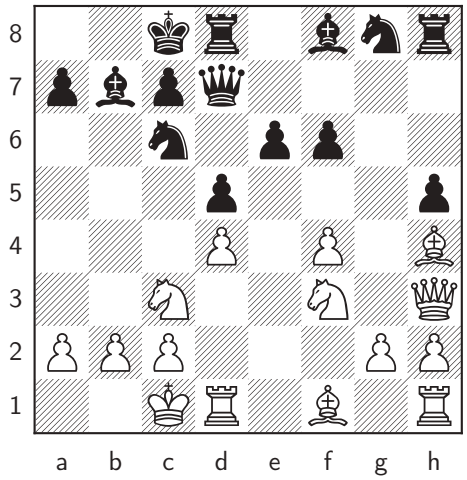

Figura B.2: Posición 2.

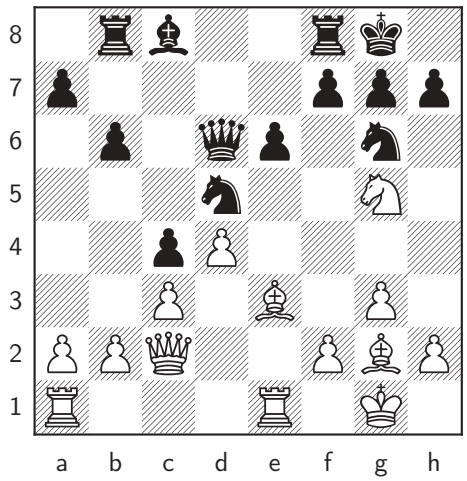

Figura B.3: Posición 3.

B.1.2. Posiciones de finales de partida

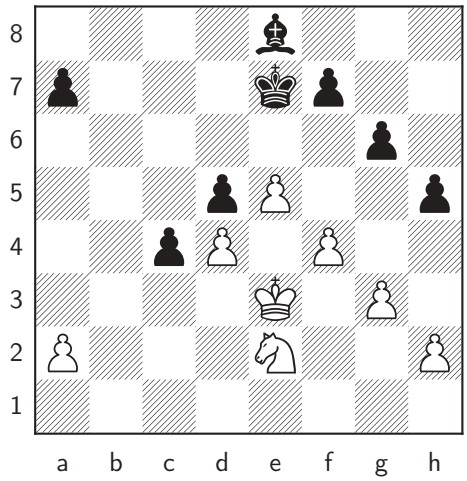

Figura B.4: Posición 4.

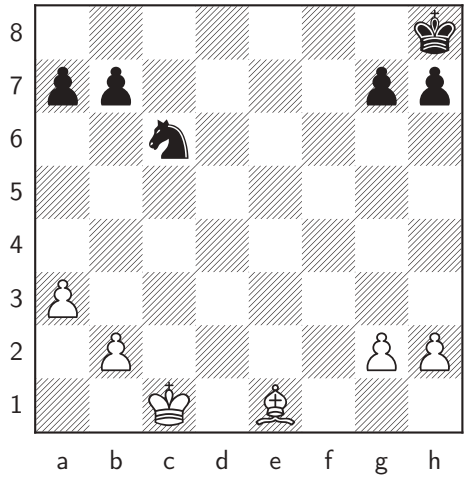

Figura B.5: Posición 5.

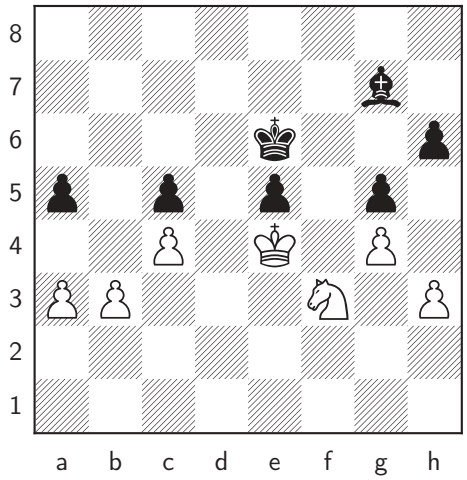

Figura B.6: Posición 6.

B.2. Resultados detallados sobre uso de heurísticas

Cuadro B.1: Posición 1.

Versión	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	197	1175	8573	66254	514148	2805215	20264864	263924160	3424958393
B	197	957	6421	22297	118020	606730	4196881	37555063	173014215
C	301	1290	9894	27714	135069	632095	4049952	30012111	145095798
D	301	1290	9003	23037	92223	345253	1481499	6501178	59154894
E	301	1290	9003	23037	121017	374201	1329825	6167497	56821400
F	301	1290	9003	22758	119463	560430	1130149	8363681	35183169
G	301	1290	5828	23232	75509	486269	1017089	7133143	29470827
H	301	1045	2528	13210	41421	373569	877639	8788686	23622040

Cuadro B.2: Posición 2.

Versión	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	101	2051	22174	158849	1021031	6515964	49243638	317806747	8254043235
B	101	1463	10231	65478	293996	2127515	11531753	73355068	367010282
C	179	1282	6539	40316	180139	1376786	6626848	28750406	311416291
D	179	1282	6539	33388	132411	563653	2496917	10231013	210803096
E	179	1282	6539	33388	131246	569540	2635595	9297615	180434674
F	179	1282	6539	32975	139066	437515	1695547	7928647	130973780
G	113	1595	7983	35215	97657	510497	2007012	5197430	119348191
H	113	1435	7919	30852	115397	508269	1674193	4183327	89798409

Cuadro B.3: Posición 3.

Versión	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	335	4035	14759	103903	638465	4999012	44403524	355875482	7822344635
B	335	3419	7036	38646	165411	1314137	7611871	71170635	352196585
C	313	2697	5925	37603	158284	1067540	7082716	65146335	279420489
D	313	2697	5559	31714	105414	593638	3076783	21437372	99754067
E	313	2697	5559	30714	109900	473935	3143294	17248258	93673184
F	313	2697	5559	31690	109114	722528	3552512	17338129	57814160
G	313	2697	3405	16572	73168	516951	3577991	15194360	40640474
H	313	2557	3305	16504	70168	500524	3014132	10342342	23784323

Cuadro B.4: Posición 4.

Versión	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	14	155	1035	9510	67982	646793	4493882	43108045	283401287
B	14	86	404	1664	6498	24650	133814	433814	3014259
C	18	68	493	3095	6638	15651	63502	217493	1078378
D	18	68	466	2612	4943	10346	29238	86892	297188
E	18	68	466	2612	4943	10346	29238	86892	297188
F	18	68	466	2612	4943	10309	29202	86568	296756
G	16	121	328	1508	4173	13126	47329	176553	712319
H	16	61	268	1159	3056	11378	41275	99563	294009

Cuadro B.5: Posición 5.

Versión	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	19	477	1589	8928	43708	248248	1297383	8034258	354235223
B	19	91	463	1521	7823	29822	146760	527061	2716398
C	21	110	483	1560	7860	29428	133585	401449	2117891
D	21	110	399	1155	3568	12361	30134	75126	171382
E	21	152	290	1318	2840	6769	23633	48178	189885
F	21	152	290	1318	2840	6769	23633	48178	189885
G	21	152	230	1114	2752	5765	20933	47323	156362
H	21	152	290	1318	2740	6569	22633	45656	178909

Cuadro B.6: Posición 6.

Versión	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
A	27	120	347	1694	17630	89808	1005328	4980692	231420881
B	26	125	412	1487	5332	21636	77218	447727	1341977
C	29	153	524	1918	7076	26812	107972	488124	1849348
D	29	153	479	1640	5020	15433	52176	120127	351278
E	29	119	214	596	1667	4864	14851	59699	153592
F	29	119	214	596	1667	4864	14851	59699	153592
G	29	153	284	324	1200	4823	15606	55088	185739
H	29	154	398	1552	1709	4064	14345	75331	156530

B.3. Resultados sobre tablas de transposición

La tabla de resultados para el esquema de reemplazo *Nunca* al no presentar en ningún caso una sola mejora, se decidió no presentarla, por no contener información relevante.

Cuadro B.7: Resultados para el esquema de remplazo *Siempre*.

Produndidad	**128K**	**256K**	**512K**	**1024K**	**2048K**
1	21	21	21	21	21
2	110	110	110	110	110
3	399	399	399	399	399
4	1155	1155	1155	1155	1155
5	3571	3570	3572	3569	3569
6	12392	12378	12372	12364	12363
7	30629	30373	30251	30192	30157
8	76989	76039	75580	75336	75199
9	190809	181463	176431	173961	172639
10	549902	503696	481215	470220	464411
11	1857389	1511308	1326428	1234178	1184411

Cuadro B.8: Resultados para el esquema de remplazo *Profundidad*.

Produndidad	**128K**	**256K**	**512K**	**1024K**	**2048K**
1	21	21	21	21	21
2	110	110	110	110	110
3	399	399	399	399	399
4	1155	1155	1155	1155	1155
5	3580	3570	3569	3573	3568
6	12395	12357	12351	12355	12351
7	30403	30239	30160	30157	30132
8	76042	75526	75302	75236	75149
9	179910	175453	173063	172305	171760
10	501198	475059	465630	462191	459645
11	2846023	1324123	1209153	1704076	1151982

Cuadro B.9: Resultados para el esquema de remplazo *Profundidad–Siempre*.

Produndidad	**128K**	**256K**	**512K**	**1024K**	**2048K**
1	21	21	21	21	21
2	110	110	110	110	110
3	399	399	399	399	399
4	1155	1155	1155	1155	1155
5	3568	3569	3568	3570	3568
6	12364	12363	12361	12363	12361
7	30177	30144	30138	30138	30134
8	75224	75162	75138	75136	75130
9	173087	172046	171589	171462	171379
10	466605	461793	459946	459391	459132
11	1230797	1174876	1150152	1140750	1137278

Apéndice C

Organización del programa *BuhoChess*

Se da una descripción de la organización del código del programa *BuhoChess*. El programa se desarrolló en C++ y se implemento mediante clases. No se dan los detalles de implantación en este apartado, ya que el código esta fuertemente comentado, sólo se pretende dar una idea de como está organizado el código, las acciones principales y de como interactuan cada una de las clases del programa. El programa está dividido en 5 archivos, tres de ellos son los módulos pincipales: generador de movimientos, función de evaluación y algoritmos de búsqueda. Los otros dos son las tablas de transposición y demás definiciones usadas por el programa. Los archivos en que se dividió el programa son:

- Chess.h
- Tablero.h
- Function.h
- Hashing.h
- Definiciones.h

En el archivo Tablero.h se encuentra «class Tablero». Contiene el módulo generador de movimientos, por lo que todos los procesos del programa de ajedrez se dan mediante esta clase. Entre sus acciones principales están:

Inicializar tablero. Se inicializa las estructuras y las variables que controlan el estado del juego en su posición inicial.

Generar estados sucesores. Dado un estado del tablero se generar todos los posibles estados sucesores.

Realizar y deshacer Movimientos Se realizan los movimientos que cambian el estado del tablero. Además, éstos se guardan en un «stack» para poder deshacerlos posteriormente.

Calcular ataques Calcula si existe un ataque entre dos casillas del tablero, normalmente es usada para calcular los ataques al rey y determinar si un movimiento es «legal».

Determinar estado del juego Determina el estado actual del juego: jaque mate o empate.

El módulo de algoritmos de búsqueda se encuentra en al archivo Chess.h, en este archivo se encuentra «class chess» y contiene los métodos para realizar las extensiones y todos los algoritmos de búsqueda. Esta es la clase principal del juego ya que funciona a manera de interfaz, recibe los movimientos realizados por el usuario y calcula los movimientos que hará la computadora. Hace uso de «class tablero» para generar los estados sucesores, los cuales son usados por los algoritmos de búsqueda. También hace uso de «class hashing» para hacer uso de las tablas de transposición durante las búsquedas y de «class fuction» para hacer uso de la función de evaluación y poder evaluar las posiciones exploradas. En esta clase se tienen las siguientes búsquedas y heurísticas.

- Búsqueda α–β.
- Búsqueda Quiscence.
- Extensiones de búsqueda.
- Búsqueda por profundidad interativa.

Durante las búsqueda se utilizan las siguientes heurísticas de ordenación de movimientos:

- Mvv/Lva.
- Heurística de historia.
- Heurística asesina.
- Tablas de transposición.

El módulo de función de evaluación se encuentra en el archivo Function.h en «class function». Éste evalúa una posición de tablero dada y regresar un valor numérico para la posición. Se tienen definidos todos los valores y estructuras utilizados por la función de evaluación que son utilizados por ésta en el mismo archivo. La clase hace uso de la clase «class tablero» para poder evaluar la posición.

Las tablas de transposición se encuentran en el archivo Hashing.h en «class hashing». En éste se tiene una estructura para las tablas «hash» y los métodos para generar las claves zobrits requeridas por las tablas. Además, se tiene todos los métodos necesarios para implementar y hacer uso de las tablas de tranposición. Esta clase es utilizada sólo por los algoritmos de búsqueda.

Las demás definiciones y estructuras utilizadas para implementar los demás heurísticas no mencionadas anteriormente se encuentran en el archivo Definiciones.h.

Bibliografía

[1] Anantharaman, T. Extension heuristics. *International Computer Chess Association Journal*, 14:47 – 65, 1991.

[2] Beal, D. F. Experiments with the null movey. *Advances in Computer Chess*, 5:65 – 79, 1989.

[3] Bell, A. G. The alpha-beta heuristic. *Games Playing with Computers*. Stanford, 1972.

[4] Berliner, H. J. Some necessary conditions for a master chess program. *The International Joint Conferences on Artificial Intelligence*. Stanford, USA, 1973.

[5] Bjornsson, Y. y T. A. Marsland. Learning search control in adversary games. *Advances in Computer Games*, 9:147–164, 2000.

[6] Björnsson, Y. y T. A. Marsland. Multi-cut alpha-beta pruning in game tree search. *Theoretical Computer Science*, 252:177 – 196, 1990.

[7] Breuker, D. M., J. Uiterwijk y H. J. van den Herik. Replacement schemes for transposition tables. *International Computer Chess Association Journal*, 17:183–193, 1994.

[8] Buro, M. Probcut: An effective selective extension of the alphabeta algorithm. *International Computer Chess Association Journal*, 18:71–76, 1995.

[9] Campbell, M. y T. S. Anantharaman. Singular extensions: Adding selectivity to brute-force searching. *AAAI Spring Symposium, Computer Game Playing*, páginas 8 – 13. 1988.

[10] Condon, J. y K. Thompson. Belle chess hardware. *Advances in Computer Chess 3*. Oxford, 1982.

[11] Donninger, C. Null move and deep search: Selective search heuristics for obtuse chess programs. *International Computer Chess Association Journal*, 5:137 – 143, 1993.

[12] Ebeling, C. *All the right moves: A VSLI architecture for chess.* Tesis Doctoral, Carnegie-Mellon University, 1986.

[13] FIDE. Fide laws of chess. http://www.fide.com/component/handbook/?id=124&view=article, 2010.

[14] Gillogly, J. *Performance Analysis of the Technology Chess Program.* Tesis Doctoral, Carnegie-Mellon University, 1987.

[15] Gillogly, J. J. The technology chess program. *Artificial Intelligence*, 3:145 – 163, 1972.

[16] Goetsch, G. y M. S. Campbell. Experiments with the null-move heuristic. *Computers, Chess, and Cognition*, páginas 159 – 168, 1990.

[17] Greenblatt, R., D. Eastlake y S. Crockerl. The greenblatt chess program. *Computing Conf. Procs.* San Francisco, 1967.

[18] Harris, L. R. The heuristic search under conditions of error. *Artificial Intelligence*, 5:217 – 234, 1974.

[19] Heinz, E. A. Darkthought goes deep. *International Computer Chess Association Journal*, 21:228 – 244, 1998.

[20] Heinz, E. A. Extended futility pruning. *International Computer Chess Association Journal*, 21:75 – 83, 1998.

[21] Heinz, E. A. *Scalable Search in Computer Chess.* Vieweg Verlag, first ed$\underline{\text{ón}}$., 2000.

[22] Hyatt, R. M., A. Gower y H. Nelson. Cray blitz. *Advances in Computer Chess 4*, páginas 8 – 18, 1985.

[23] Junghanns, A., J. Schaeffer, M. Brockington, Y. Bjornsson y T. Marsland. Diminishing returns for additional search in chess. *Advances in Computer Chess*, 8:53 – 67,, 1997.

[24] Kaindl, H. Dynamic control of the quiescence search in computer chess. *Cybernetics and Systems Research*, páginas 973 – 977, 1982.

[25] Knuth, D. E. *The Art of Computer Programming.* Addison - Wesley, 1973.

[26] Louguet, F. Lct ii. http://chessprogramming.wikispaces.com/LCT+II, 2010.

[27] Marsland, T. A. Relative efficiency of alpha-beta implementations. *International Joint Conference on Artificial Intel ligence*. Karlsruhe, Germany, 1983.

[28] Marsland, T. A. Evaluation-function factors. *International Computer Chess Association Journal*, 8:47–57, 1985.

[29] Marsland, T. A. A review of game-tree pruning. *International Computer Chess Association Journal*, 9:3 – 9, 1986.

[30] Marsland, T. A. y M. S. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14:533–551, 1982.

[31] Moreland, D. 0x88 move generation. http://web.archive.org/web/20070716111804/www.brucemo.com/compchess/programming/0x88.htm, 2010.

[32] Neumann, J. V. y O. Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[33] Reinefeld, A. An improvement to the scout search tree algorithm. *International Computer Chess Association Journal*, 4:4 – 14, 1983.

[34] Richard, E. K. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[35] Schaeffer, J. The history heuristic. *International Computer Chess Association Journal*, 41:16–19, 1983.

[36] Selim, G. A. y M. Monroe. The principal continuation and the killer heuristic. *Proceedings of the 1977 annual conference*. New York, NY, USA, 1977.

[37] Shannon, C. E. Programing a computer for playing chess. *philosophical magazine*, 41:29 – 48, 1950.

[38] Slagle, J. H. y J. K. Dixon. Experiments with some programs that search game trees. *Journal of the ACM*, 16:189–207, 1969.

[39] Slate, D. y L. Atkin. Chess skill in man and machine. *The Northwestern University Chess Program*, 4:29 – 48, 1977.

[40] Slate, D. J. y L. Atkin. Chess 4.5 the northwestern university chess program. *Chess Skill in Man and Machine*, páginas 82 – 118, 1977.

[41] Slater, E. Statistics for the chess computer and the factor of mobility. *Proceedings of the Symposium on Information Theory*. London, 1988.

[42] Stuart, R. y J. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2003.

[43] Thompson, K. Computer chess strength. *Advances in Computer Chess*, 3:55 – 56, 1982.

[44] Turing, A. M. Digital computer applied to games. *Faster than Thounght: a sumposium on digital computing Machines.* Londres, 1953.

[45] Zobrist, A. L. *A New Hashing Method with Applications for Game Playing.* Tesis Doctoral, Computer Sciences Dept. Univ. of Wisconsin, 1990.