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;

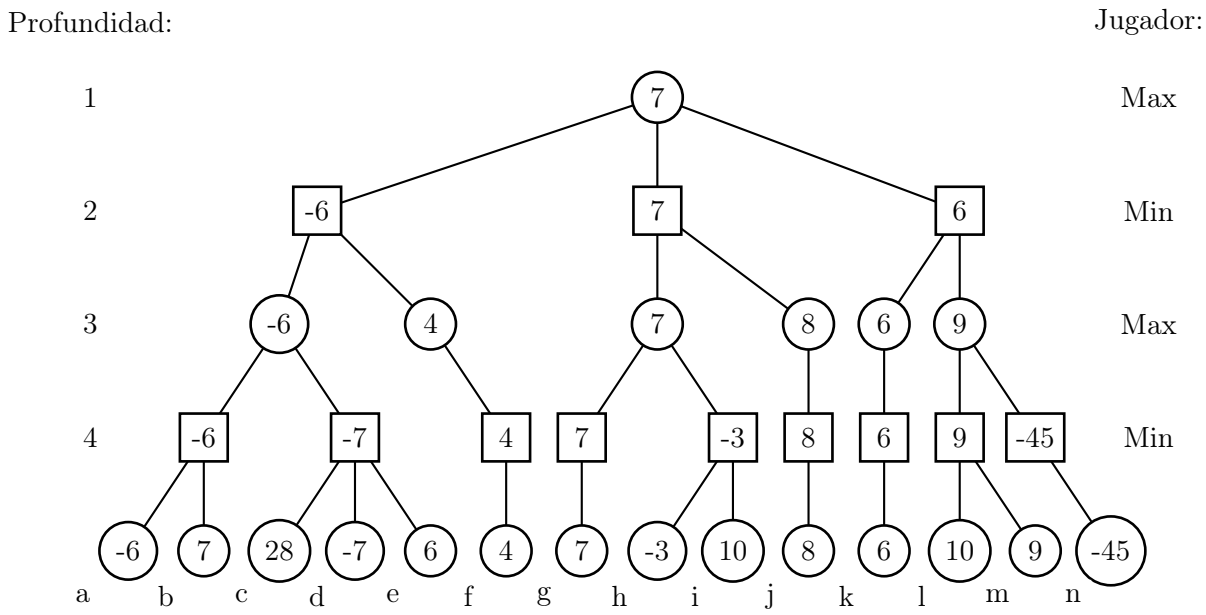

Figura 3.1: Árbol de Búsqueda *Minimax*

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*.

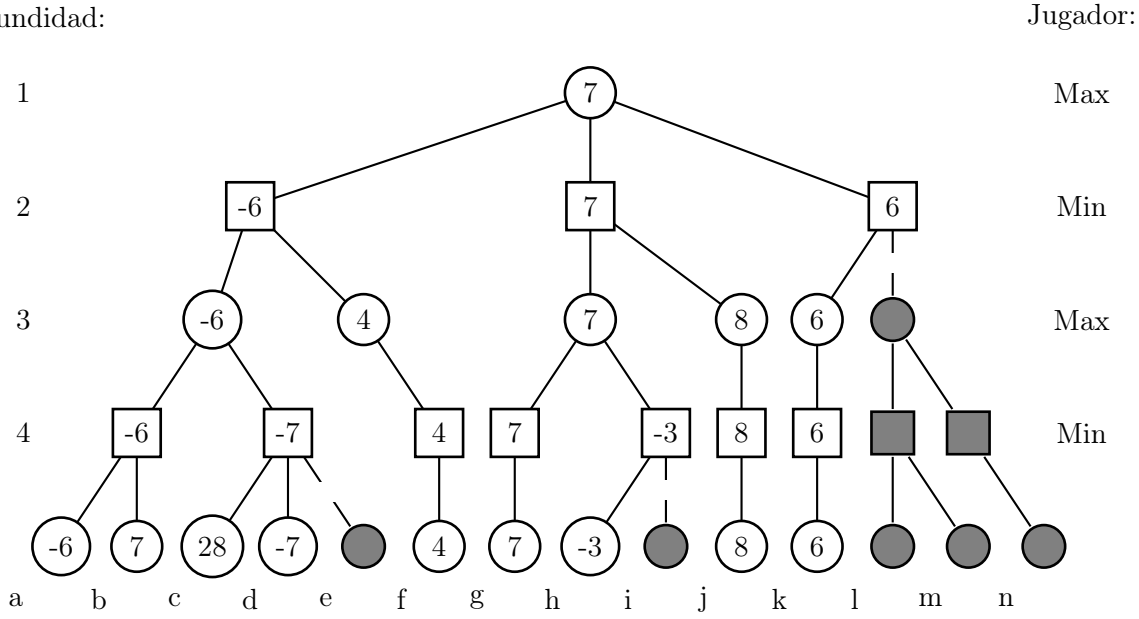

Figura 3.2: Árbol de Búsqueda $\alpha - \beta$.

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`)`

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 una estrategia establecida
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**
    regresar β
  fin si
  si $valor > \alpha$ **entonces**
    $\alpha \leftarrow valor$
  fin si
fin para
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.

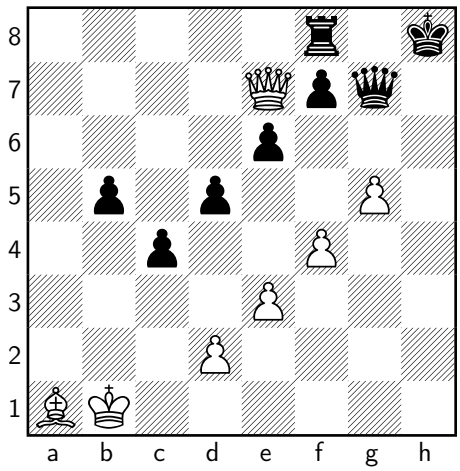

Figura 3.3: Ejemplo del efecto horizonte.

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.