Estudio de métodos de decisión en juegos con adversario

Grado en Matemáticas
Trabajo de fin de grado

UNIVERSIDAD
COMPLUTENSE
MADRID

Mario Carrillo Redondo
Tutor: Fernando Rubio Diez
Departamento de Sistemas Informáticos y Computación

Julio 2023

Índice general

Resumen

Desde hace décadas se han desarrollado interesantes métodos de decisión para juegos con adversario, siendo bien conocidos métodos como el minimax (y sus múltiples variantes y mejoras) o los árboles de búsqueda Monte Carlo. El objetivo del presente trabajo será estudiar un amplio repertorio de dichos métodos, implementarlos y analizar su eficacia para resolver juegos concretos.

Palabras clave

Árboles de juego, juegos con adversario, *minimax*, poda alfa-beta, búsqueda de la variante principal, *quiescence searh*, árboles de búsqueda de Monte Carlo.

Abstract

For decades, interesting decision methods have been developed for adversarial games, with well-known methods such as *minimax* (and its multiple variants and improvements) or Monte Carlo tree search. The objective of this work will be to study a wide repertoire of these methods, implement them, and analyze their effectiveness in solving specific games.

Key words

Game tree, adversarial games, *minimax*, alpha–beta pruning, principal variation search, quiescence searh, Monte Carlo tree search.

Capítulo 1

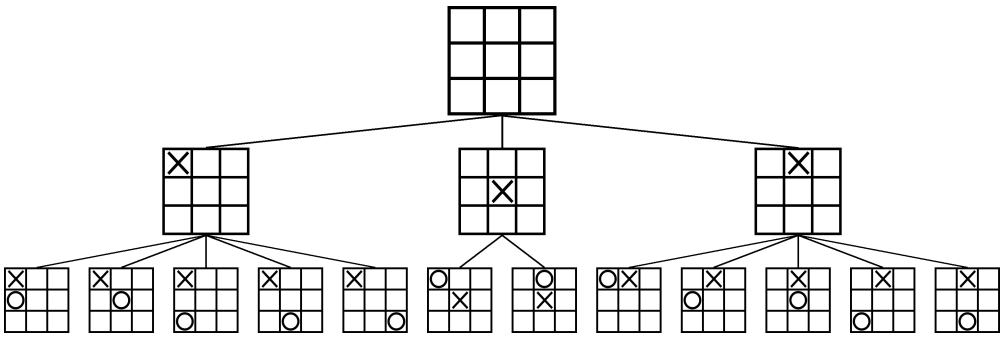
Introducción

El objetivo de este trabajo es analizar métodos generales para dada una posición en un juego encontrar una siguiente jugada prometedora. Para ello analizaremos diferentes algoritmos generales como el minimax y el MCTS, y veremos el resultado de aplicarlos a diferentes juegos. La representación de los juegos la haremos a traves de una estructura de datos: los árboles de juego.

Definición 1.1 (Árbol de juego)**.** Un árbol de juego es un árbol en el que cada nodo representa un estado del juego. Las transiciones de cada nodo a cada uno de sus hijos (de existir) son los movimientos. El nodo raíz del árbol es el estado inicial del juego. Los nodos terminales (es decir, los que no tienen hijos) son los estados en los que el juego termina y en ellos se puede evaluar el resultado del juego.

Un árbol de juego es una estructura de datos recursiva, por tanto después de elegir un movimiento acabas en un nodo hijo que es el nodo raíz de su propio subárbol de juego. Por tanto, podemos entender una partida como una secuencia de decisiones de movimientos representadas cada vez por un árbol de juego diferente. La mayoria de las veces en la práctica no tenemos que recordar el camino hasta el estado actual.

Ejemplo 1.2. El *tic-tac-toe* (o tres en raya) es un juego de dos jugadores los cuales se turnan marcando espacios en blanco de una matriz 3x3 con X o O. Un jugador gana si consigue colocar tres de sus marcas en una horizontal, una vertical o una diagonal. Estos son los tres primeros niveles del árbol de juego del *tic-tac-toe* eliminando posiciones simétricas:



Otro concepto importante, ya que está relacionado con la complejidad de los algoritmos que vamos a estudiar, es el factor de ramificación.

Definición 1.3 (Factor de ramificación)**.** Dado un árbol de juego se denomina factor de ramificación al número de nodos hijos en cada nodo. Si este valor no es uniforme, se puede

calcular el factor de ramificación medio. Denotando por b al factor de ramificación medio sería:

$$b = \frac{n-1}{n_h}$$

donde n = Número de nodos del árbol y n_h = Número de nodos del árbol con hijos

Ejemplo 1.4. En el tic-tac-toe, el factor de ramificación varía para cada nodo pero como el árbol de juego es pequeño se puede calcular el factor de ramificación medio por fuerza bruta. El número de nodos del árbol de juego es de 549946 y el número de nodos del árbol con hijos es de 294778, por tanto:

$$b = \frac{549946-1}{294778} = 1{,}8656243003209194$$

La complejidad del algoritmo *minimax* que veremos a continuación es de $\Theta(b^d)$ donde b es el factor de ramificación del juego y d la profundidad a la que queremos buscar. En variantes del *minimax* como la poda alfa beta, la complejidad del algoritmo mejora si podemos ordenar los nodos correctamente, ya que esto nos permite podar más ramas.

Capítulo 2

Minimax y sus variantes

2.1. *Minimax*

El algoritmo *minimax* es un algoritmo que se utiliza en teoría de juegos para encontrar el movimiento óptimo para un jugador, suponiendo que el rival también juega de manera óptima. Es usado comúnmente en juegos de dos jugadores por turnos como el *tic-tac-toe*, *backgammon*, ajedrez, etc.

En el *minimax* los dos jugadores son llamados 'max' y 'min'. El jugador 'max' trata de conseguir la mayor puntuación posible mientras que el 'min' trata de hacer lo contrario, conseguir la menor puntuación posible.

La idea del algoritmo es la siguiente, dada una posición en un juego, vamos a examinar todos los posibles finales a los que podemos llegar desde la posición actual y vamos a asignarles un valor. Usualmente, se le dará a una posición final ganada para 'max' el valor ∞, a una posición final empatada el valor 0 y a una posición final ganada para 'min' el valor $-\infty$. Después, empezando desde las posiciones finales vamos a propagar los resultados de abajo a arriba hasta la posición actual de la siguiente manera: si estamos en un nodo en el que le toca jugar a 'min' haremos la jugada que menor valor tenga entre sus hijos (es decir, 'min' hace su mejor jugada posible) y si estamos en un nodo en el que le toca jugar a 'max' haremos la jugada que mayor valor tenga entre sus hijos (es decir, 'max' hace su mejor jugada posible).

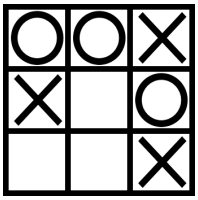
En la práctica, una búsqueda tan exhaustiva hasta las posiciones finales es imposible para la mayoría de juegos. En su lugar lo que se hace es limitar la profundidad a la que se realiza la búsqueda. Pero para poder limitar la profundidad a la que realizamos la búsqueda necesitamos poder evaluar posiciones no finales (ya que no estaría garantizado que todas las posiciones a las que llegamos sean finales). Para ello necesitamos una función heurística, es decir, una función que dada una posición cualquiera nos devuelva una evaluación del tablero sin mirar pasos más alla. Una heurística típica es la que se utiliza en ajedrez, dándole un valor a cada pieza y sumando esos valores (y restando los valores del oponente) donde una dama vale 9, una torre 5, un caballo o un alfil 3 y un peón 1.

Algoritmo 2.1. El pseudocódigo puede verse en el algoritmo 1 y para usarlo haremos una llamada inicial con $nodo = posicionActual$, $profundidad$ la que queramos y $etiqueta =$ "max" en nuestro turno.

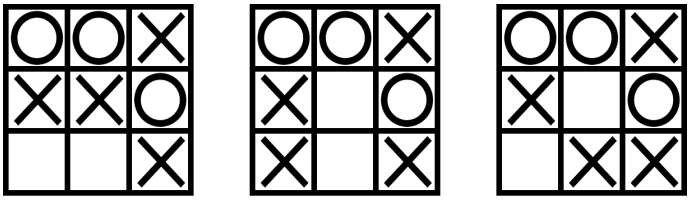
Algoritmo 1: *Minimax*

```
1  minimax (nodo, profundidad, etiqueta)
2      if profundidad = 0 or nodo es final then
3          return heuristica(nodo, etiqueta)
4      else
5          if etiqueta = "max" then
6              valor ← −∞;
7              foreach hijo de nodo do
8                  valor ← máx{valor, minimax(hijo, profundidad − 1, "min")};
9              return valor;
10         else
11             valor ← ∞;
12             foreach hijo de nodo do
13                 valor ← mín{valor, minimax(hijo, profundidad − 1, "max")};
14             return valor;
```

Ejemplo 2.2 (*Minimax* aplicado al *tic-tac-toe*)**.** Supongamos que en una partida de *tic-tac-toe* es nuestro turno, jugamos X y la posición actual es la siguiente:



Queremos saber si estamos ganando en esta posición asi que utilizamos el algoritmo *minimax* con $profundidad = \infty$ para llegar a las posiciones finales. Los hijos de nuestra posición tendrían la etiqueta 'min' y serían los siguientes:



Como el nodo inicial es 'max' ahora tendríamos que aplicar el algoritmo para cada hijo y quedarnos con el máximo de los tres resultados. Como se puede ver en el árbol 2.1, los hijos 1 y 3 devuelven 0 al aplicar el algoritmo mientras que el hijo 2 devuelve ∞. Por tanto, el algoritmo devuelve ∞ para el nodo inicial, lo que significa que jugando bien la posición está ganada.

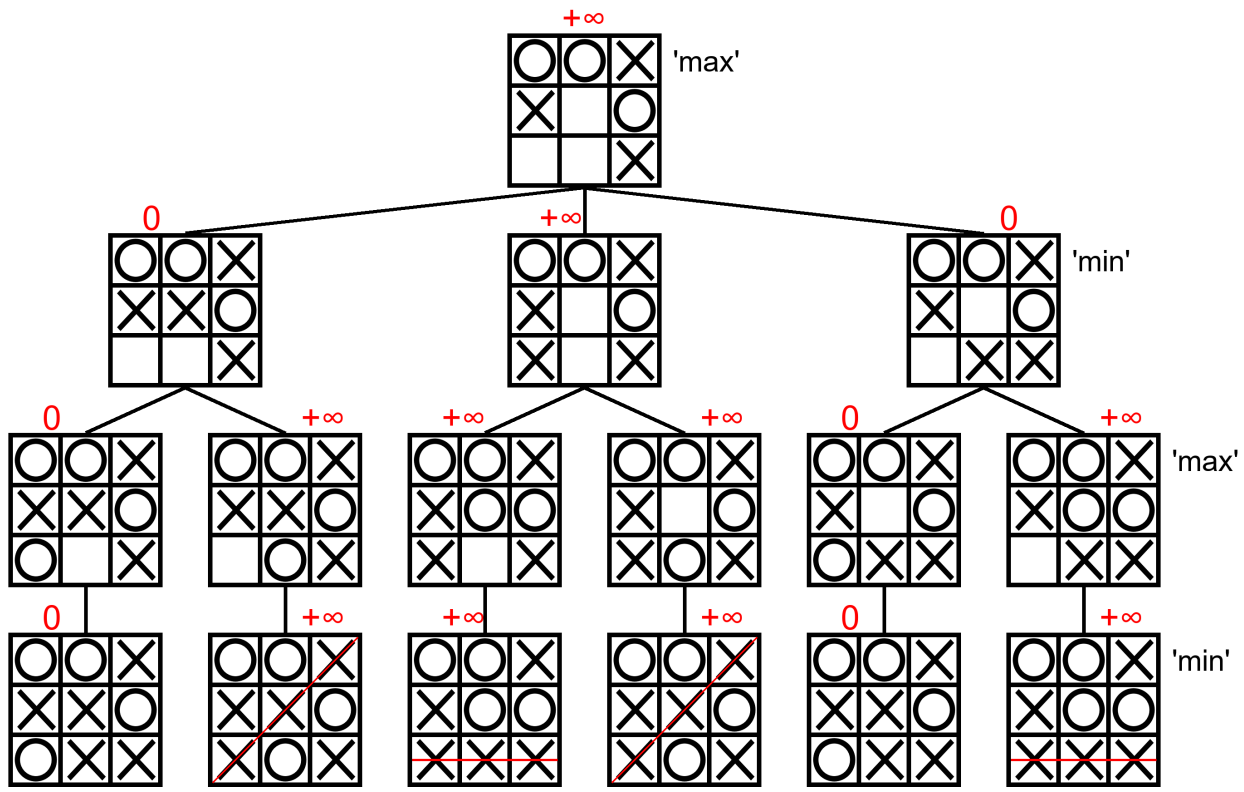


Figura 2.1: Árbol *minimax* completo

Observación 2.3. Podemos utilizar el *minimax* para determinar si en un juego existe una estrategia ganadora para el primer jugador evaluando el árbol de juego desde la posición inicial y con $profundidad = \infty$. Esto quiere decir que podemos determinar si el primer jugador tiene victoria asegurada haciendo un juego perfecto. En el caso del *tic-tac-toe*, desde la posición inicial el algoritmo devuelve 0, por lo que se concluye que no existe una estrategia ganadora.

2.2. Poda alfa-beta

El algoritmo de poda alfa-beta es una mejora al algoritmo *minimax*. El problema con el algoritmo *minimax* es que el número de posiciones que tiene que examinar es exponencial en el número de movimientos. Aunque es imposible eliminar el exponente completamente podemos reducirlo a la mitad. Es posible devolver la misma decisión que el algoritmo *minimax* sin mirar a cada nodo del árbol de juego. La idea de la poda nos permite eliminar posibilidades sin tener que examinarlas a partir de cortar ramas que no van a influir en la decisión final.

La poda alfa-beta se puede aplicar a la profundidad que queramos y muchas veces nos permite cortar subárboles enteros, no solo hojas. La idea del algoritmo es la siguiente:

1. Consideremos un nodo n en el árbol de juego el cual puede ser alcanzado.

2. Si hay una mejor opción m ya sea en el padre del nodo n o en cualquier nodo por encima en el árbol de juego, n nunca será alcanzado.

3. Cuando tenemos la suficiente información sobre n para llegar a esta conclusión, lo podemos podar.

El algoritmo de poda alfa-beta recibe su nombre por los siguientes parámetros que se utilizan en el mismo:

- α = El valor de la mejor decisión que hemos encontrado hasta ahora en cualquier punto de decisión en el camino para 'max'
- β = El valor de la mejor decisión que hemos encontrado hasta ahora en cualquier punto de decisión en el camino para 'min'

Más adelante veremos con un ejemplo como el algoritmo mejora con una buena ordenación de los hijos.

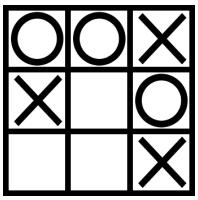
Algoritmo 2.4. El algoritmo en pseudocódigo sería el siguiente:

Algoritmo 2: Poda alfa-beta

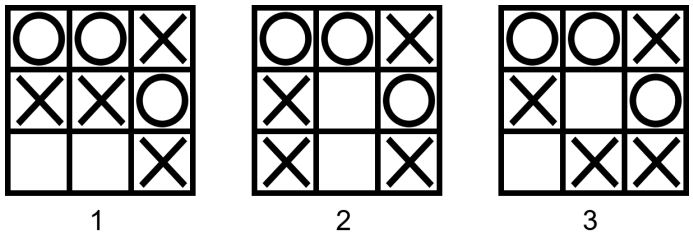
```
1  alfabeta (nodo, profundidad, α, β, etiqueta)
2      if profundidad = 0 or nodo es final then
3          return heuristica(nodo, etiqueta)
4      else
5          if etiqueta = "max" then
6              valor ← −∞;
7              foreach hijo de nodo do
8                  valor ← máx{valor, alfabeta(hijo, profundidad − 1, α, β, "min")};
9                  if valor > β then
10                     break; // Poda beta
11                 α ← máx{α, valor};
12             return valor;
13         else
14             valor ← ∞;
15             foreach hijo de nodo do
16                 valor ← mín{valor, alfabeta(hijo, profundidad − 1, α, β, "max")};
17                 if valor < α then
18                     break; // Poda alfa
19                 β ← mín{β, valor};
20             return valor;
```

y para usarlo haremos una llamada inicial con $nodo = posicionActual$, $profundidad$ la que queramos, $\alpha = -\infty$, $\beta = \infty$ y $etiqueta =$ "max" en nuestro turno.

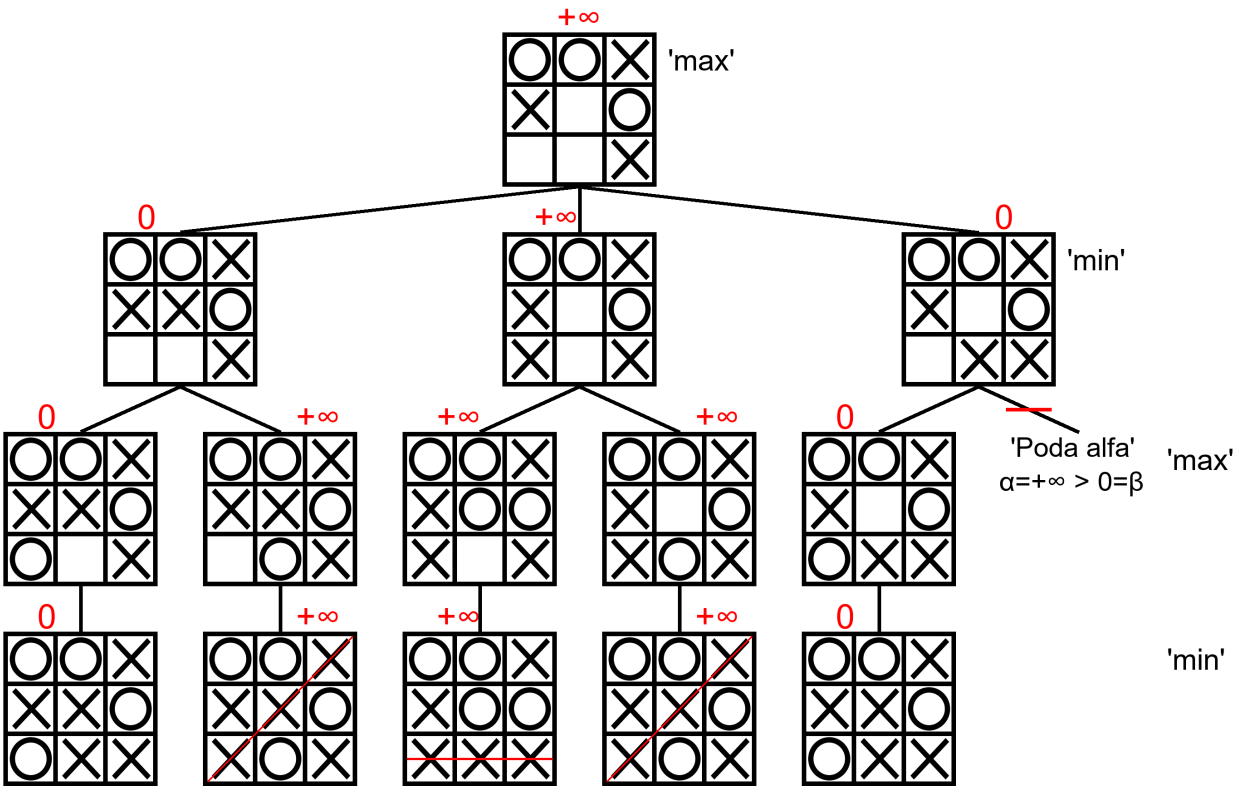
Ejemplo 2.5 (Poda alfa-beta aplicada al *tic-tac-toe*)**.** Vamos a comprobar que la poda alfa-beta toma la misma decisión que el *minimax* pero más rápido usando la misma posición que en el ejemplo del *minimax*. Nos encontrábamos en la siguiente posición, era nuestro turno y jugábamos X:



Al igual que con el *minimax* vamos a usar $profundidad = \infty$ para llegar a las posiciones finales e inicializamos con $\alpha = -\infty$ y $\beta = \infty$. Los hijos de nuestra posición tendrían la etiqueta 'min' y serían los siguientes:

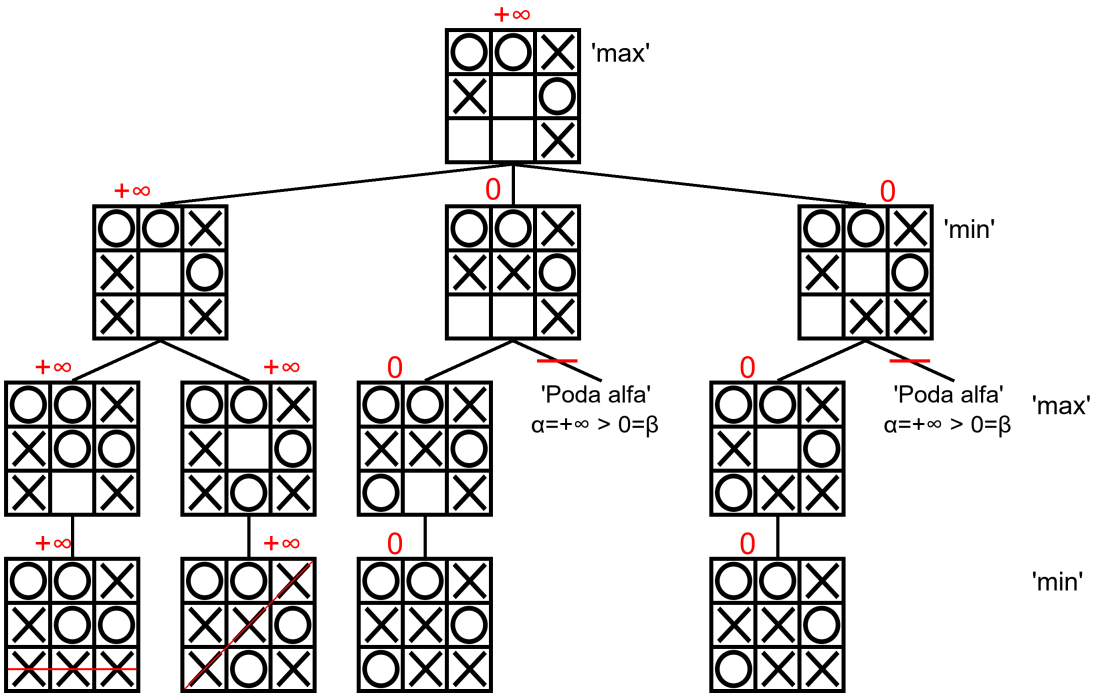


Empezamos aplicando el algoritmo al hijo 1. El jugador 'min' puede hacer una jugada que empata por lo que pasa a ser $\alpha = 0$ y β se mantiene igual (a pesar de haber cambiado momentáneamente su valor al entrar por el primer subhijo). Con estos valores aplicamos el algoritmo al hijo 2. El jugador 'min' no puede encontrar una jugada mejor que la derrota por lo que se actualiza $\alpha = \infty$ y β se mantiene igual. Al aplicar el algoritmo al último hijo, $\alpha = \infty$ por lo que al actualizar $\beta = 0$ en el primer subhijo se da que $\alpha > \beta$ y se hace la poda alfa. Este sería el árbol resultante:

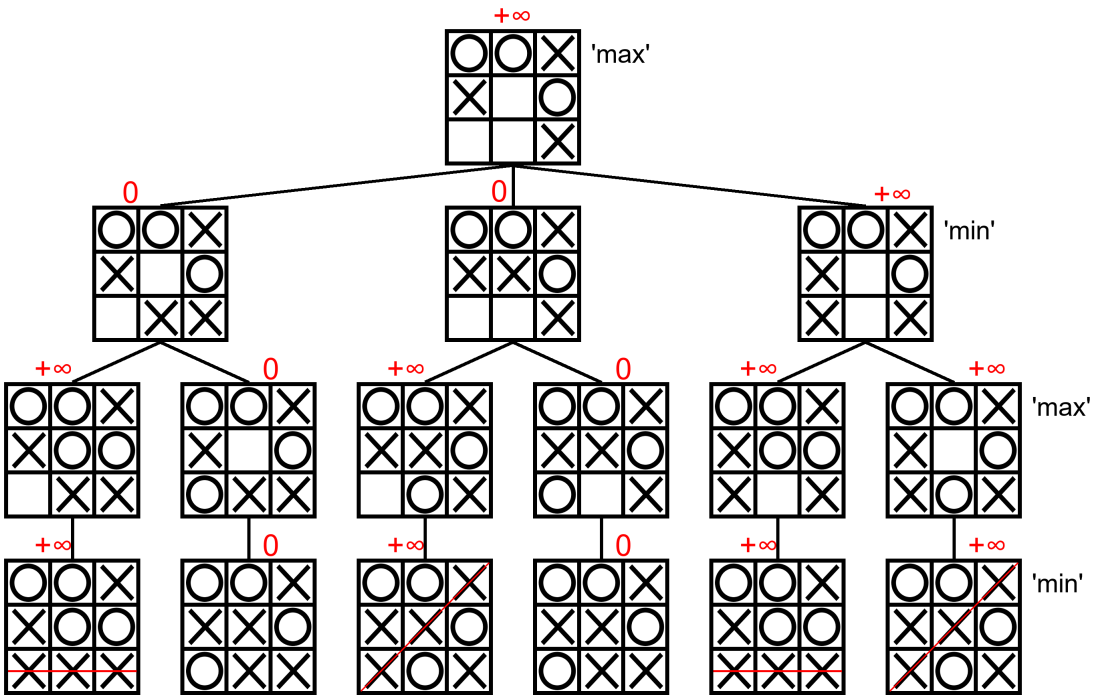


Como se puede ver en los árboles de arriba, la decisión al igual que para el *minimax* es hacer la jugada del hijo 2 para el jugador 'max' pero al usar el algoritmo de poda alfa-beta nos ahorramos calcular una rama.

Observación 2.6. Como ya mencionamos en la introducción, el orden en el que exploremos los hijos influye en el número de podas que haremos. Para optimizar la búsqueda deberíamos ordenar los nodos en orden creciente para los hijos de nodos 'min' y en orden decreciente para los hijos de nodos 'max' en función de una heurística. Como vimos, en el ejemplo se hace 1 poda. Si ordenamos los nodos de forma óptima, es decir de la siguiente manera, hacemos 2 podas.



Mientras que si lo hacemos de la forma menos óptima, es decir de la siguiente manera, hacemos 0 podas.



2.3. Búsqueda de la variante principal

La búsqueda de la variante principal es una variante del algoritmo *minimax* que puede llegar a ser más rápida que la poda alfa-beta. Cualquier nodo que sea podado por la poda alfa-beta no será examinado por la búsqueda de la variante principal, pero es necesario una buena ordenación de los nodos para conseguir una mejora.

La búsqueda de la variante principal funciona mejor cuando las jugadas están bien ordenadas. En la práctica, el orden de los movimientos se suele determinar a través de búsquedas previas menos profundas. El algoritmo hace más podas que la poda alfa-beta a partir de suponer que el primer nodo examinado es el mejor (es decir, a partir de suponer que es la variante principal). Después, comprueba si esto era cierto a partir de buscar en los nodos restantes con una ventana nula (con alfa y beta iguales) lo cual es más rápido que usar la ventana normal de la poda alfa-beta. Si la prueba falla, entonces el primer nodo no era la variante principal, y la búsqueda continúa como una poda alfa-beta normal. Es por ello que la búsqueda de la variante principal funciona mejor con una buena ordenación. Con una ordenación aleatoria de los nodos la búsqueda de la variante principal tardará más tiempo que la poda alfa-beta ya que aunque no vaya a examinar nodos podados por la poda alfa-beta sí que volverá a buscar en muchos nodos.

Algoritmo 2.7. El algoritmo en pseudocódigo sería el siguiente:

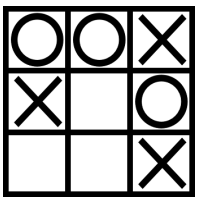
Algoritmo 3: Búsqueda de la variante principal

```
1  bvp (nodo, profundidad, α, β, lado)
2      if profundidad = 0 or nodo es final then
3          return lado · heuristica(nodo, lado)
4      else
5          foreach hijo de nodo do
6              if hijo es el primer hijo then
7                  valor ← −bvp(hijo, profundidad − 1, −β, −α, −lado);
8              else
9                  valor ← −bvp(hijo, profundidad − 1, −α − 1, −α, −lado);
                    // Búsqueda con la ventana nula
10                 if α < valor < β then
11                     valor ← −bvp(hijo, profundidad − 1, −β, −valor, −lado); // Si
                        falló la ventana nula, hace una nueva búsqueda
12             α ← máx{α, valor};
13             if α ≥ β then
14                 break; // Poda beta
15     return α;
```

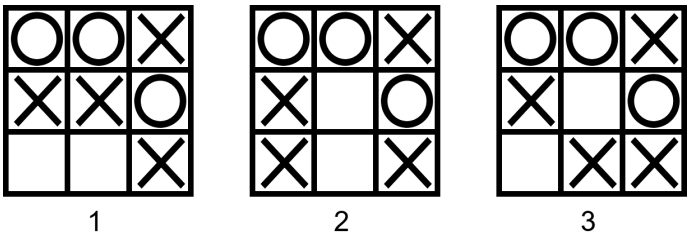
y para usarlo haremos una llamada inicial con $nodo = posicionActual$, $profundidad$ la que queramos, $\alpha = -\infty$, $\beta = \infty$ y $lado = 1$ en nuestro turno.

Una diferencia de este algoritmo con respecto al que usamos para la poda alfa-beta es que el corte se produce si $\alpha >= \beta$ mientras que en el que usamos para la poda alfa-beta el corte se producía únicamente si $\alpha > \beta$.

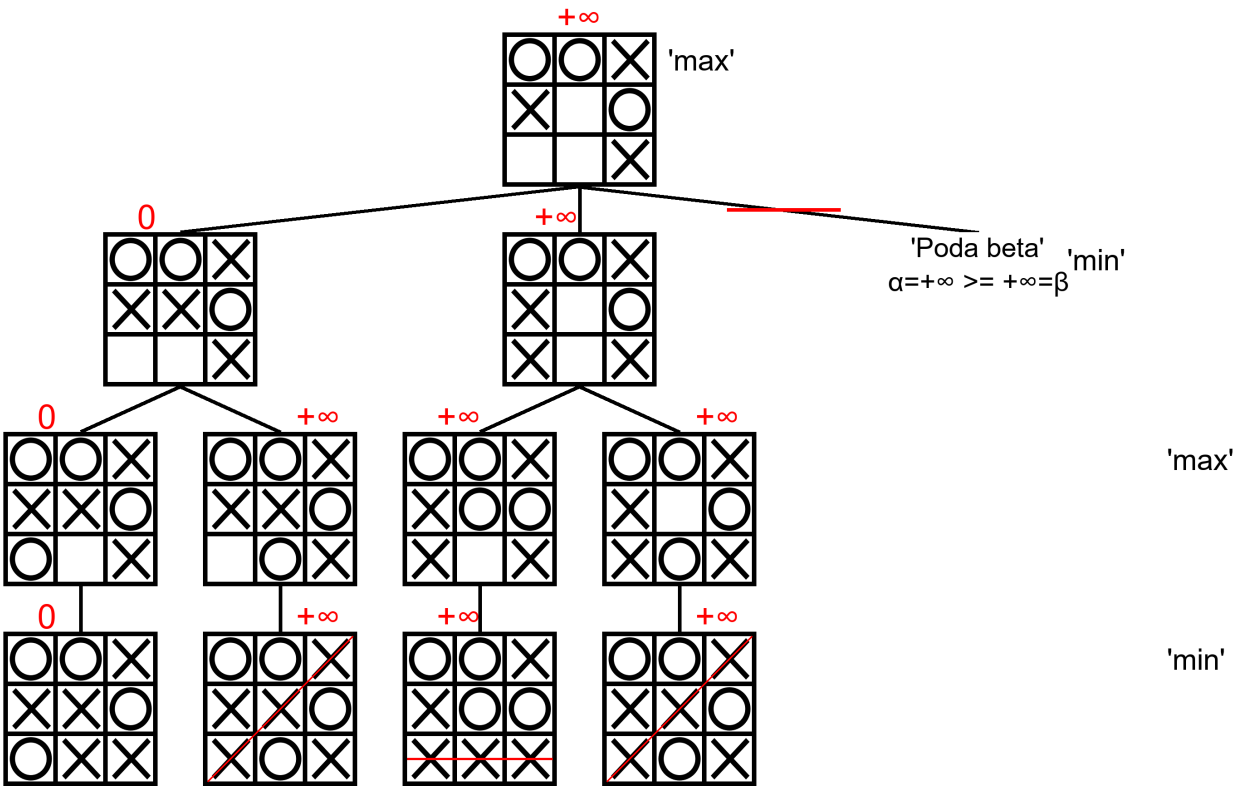
Ejemplo 2.8 (Búsqueda de la variante principal aplicada al *tic-tac-toe*)**.** Vamos a comprobar que la búsqueda de la variante principal toma la misma decisión que el *minimax* pero más rápido usando la misma posición que en los ejemplos anteriores. También, vamos a ver qué diferencias hay en la forma de decidir de este algoritmo y la poda alfa-beta. Nos encontrábamos en la siguiente posición, era nuestro turno y jugábamos X:



Al igual que con la poda alfa-beta vamos a usar $profundidad = \infty$ para llegar a las posiciones finales y inicializamos con $\alpha = -\infty$ y $\beta = \infty$. Los hijos de nuestra posición tendrían la etiqueta 'min' y serían los siguientes:

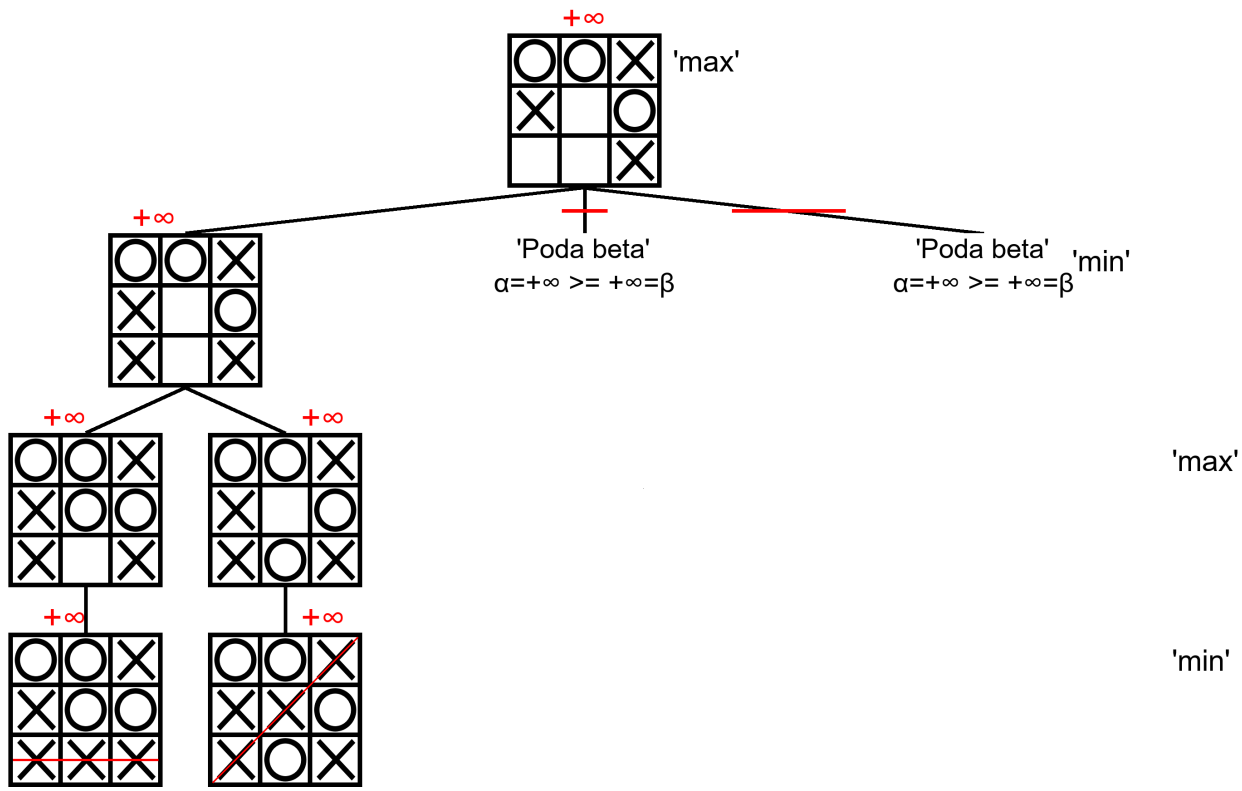


Empezamos aplicando el algoritmo al hijo 1, el cual suponemos que es el mejor. El algoritmo nos devuelve 0 y entra al hijo 2, para el cual supone que ya no es el mejor y por tanto se realiza la búsqueda con la ventana nula. En este caso, el hijo 2 es el mejor por lo que la ventana nula falla y tenemos que recalcular este hijo con la ventana (α, β). Al recalcular vemos que este hijo sube el valor de alfa a $\alpha = \infty$ y como se cumple que $\alpha = \beta$ se poda el hijo 3 y hemos acabado. El árbol sería el siguiente:



Como se puede ver en los árboles de arriba, la decisión al igual que para el *minimax* y la poda alfa-beta es hacer la jugada del hijo 2 para el jugador 'max'. Al usar la búsqueda de la variante principal nos ahorramos calcular una rama entera pero por no tener una buena ordenación de los hijos tenemos que recalcular una rama entera también.

Observación 2.9. Si ordenamos los nodos de manera óptima, es decir como en la observación que vimos para la poda alfa-beta, este algoritmo nos ofrece la solución más rápida posible. Esto se debe más al hecho de que estamos siendo laxos con los cortes y cortando si $\alpha = \beta$ también más que a la idea de la ventana nula, pero sirve para ver la mejora que supone cambiar esa restricción. El árbol sería el siguiente:



2.4. *Quiescence search*

Cuando fijamos una profundidad a la que buscar desde un nodo en un árbol de juego, las amenazas y las oportunidades a más profundidad de la fijada no son detectadas. Esto puede dar lugar a que los algoritmos se encuentren haciendo movimientos de espera hasta que con la profundidad que tienen fijada puedan ver las amenazas, y en ocasiones puede ocurrir que ya sea demasiado tarde. El algoritmo de *quiescence search* intenta mitigar este error extendiendo la profundidad de búsqueda en posiciones "volátiles" en las cuales el valor de la función heurística pueda tener grandes fluctuaciones entre los diferentes movimientos.

Este es el algoritmo que simula mejor el juego de un humano. Los jugadores humanos tienen la intuición para decidir si abandonar un movimiento que parece malo o si seguir buscando en un movimiento que parece bueno a mayor profundidad. La *quiescence search* intenta simular este comportamiento a través de extender la búsqueda en posiciones "volátiles" más allá que en posiciones "tranquilas" para asegurar que no hay trampas ocultas y obtener una mejor estimación de las posiciones.

Cualquier criterio puede ser usado para distinguir las posiciones "tranquilas" de las posiciones "volátiles". Uno de los criterios habituales es ver si existen movimientos en la posición que cambian drásticamente la evaluación de la posición. La *quiescence search* no acaba mientras la posición siga siendo volátil de acuerdo al criterio. Para lidiar con esto y que la búsqueda termine se suele restringir la búsqueda hasta movimientos que lidian directamente con la amenaza, como las capturas en el ajedrez. En juegos altamente "inestables" como el

Go o el *reversi*, se necesitaría más tiempo para poder aplicar la *quiescence search*.

Algoritmo 2.10. El algoritmo en pseudocódigo sería el siguiente:

Algoritmo 4: *Quiescence search*

```
1  quiescenceSearch (nodo, profundidad, etiqueta)
2      if profundidad = 0 or nodo es final or nodo es tranquilo then
3          return heuristica(nodo, etiqueta)
4      else
5          (Buscar recursivamente en los hijos del nodo usando quiescenceSearch)
```

y para usarlo haremos una llamada inicial con $nodo = posicionActual$, $profundidad$ la que queramos y $etiqueta =$ "max" en nuestro turno. La parte recursiva del algoritmo no está especificada ya que la *quiescence search* puede usarse para cualquiera de los algoritmos ya descritos, solo proporciona una mejora.

Observación 2.11. El algoritmo descrito arriba tiene una particularidad, si una posición parece "tranquila" deja de profundizar en ella inmediatamente, sin llegar a la profundidad que hemos fijado de entrada. Otra forma en la que podríamos usar la *quiescence search* es buscando hasta la profundidad que hemos fijado y extendiendo la búsqueda en los nodos "volátiles" al final hasta una segunda profundidad de búsqueda razonable. Este algoritmo en pseudocódigo sería el siguiente:

Algoritmo 5: *Normal search*

```
1  normalSearch (nodo, profundidad, profundidad2, etiqueta)
2      if nodo es final then
3          return heuristica(nodo, etiqueta)
4      else if profundidad = 0 then
5          if nodo es tranquilo then
6              return heuristica(nodo, etiqueta)
7          else
8              return quiescenceSearch(nodo, profundidad2, etiqueta)
9      else
10         (Buscar recursivamente en los hijos del nodo usando normalSearch)
```

Al igual que pasaba con la *quiescence search*, puede usarse para cualquiera de los algoritmos ya descritos para la búsqueda recursiva de los hijos en este algoritmo.

Ejemplo 2.12 (*Quiescence search* aplicada al ajedrez)**.** Supongamos que estamos aplicando el algoritmo *minimax* con una función heurística que solo tiene en cuenta el valor de las piezas en una partida de ajedrez y hemos llegado a la posición 2.2 con $profundidad = 1$ siendo el turno de las blancas.

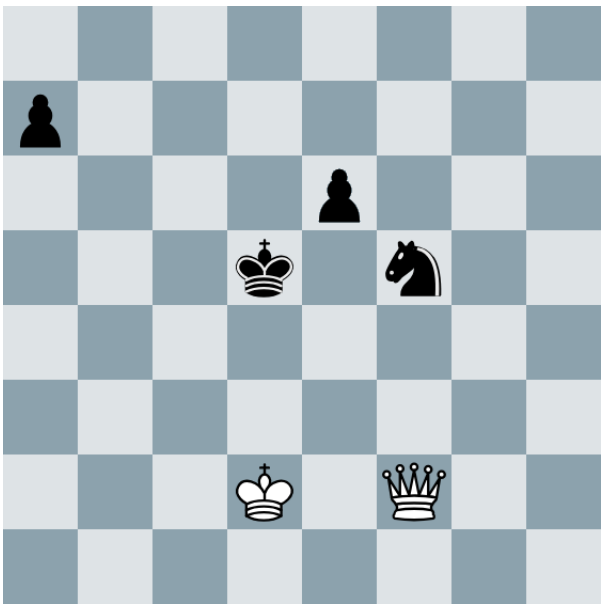


Figura 2.2: Posición 0

De entre las jugadas posibles la que deja a las blancas con mejor ventaja de material en el turno siguiente es la posición 2.3.

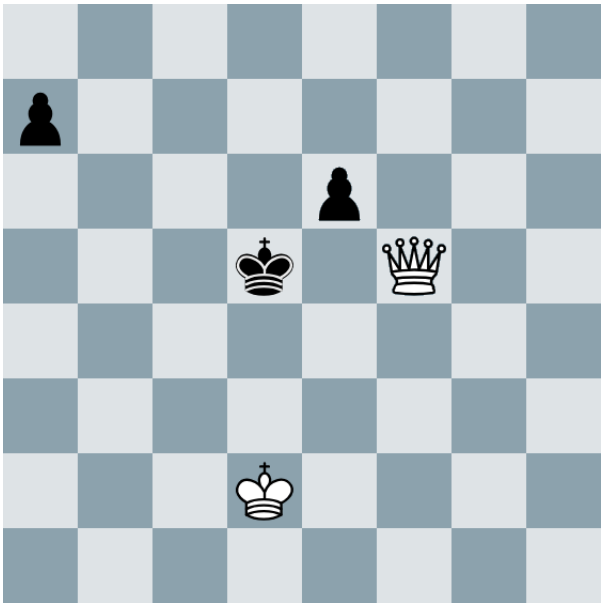


Figura 2.3: Posición 1

Como estábamos analizando la posición anterior con $profundidad = 1$ el algoritmo haría esa jugada. Sin embargo, la posición resultante de comer el caballo está perdida ya que el intercambio de piezas no acaba ahí. En la jugada siguiente el jugador negro puede capturar la dama con el peón, llegando a la posición 2.4.

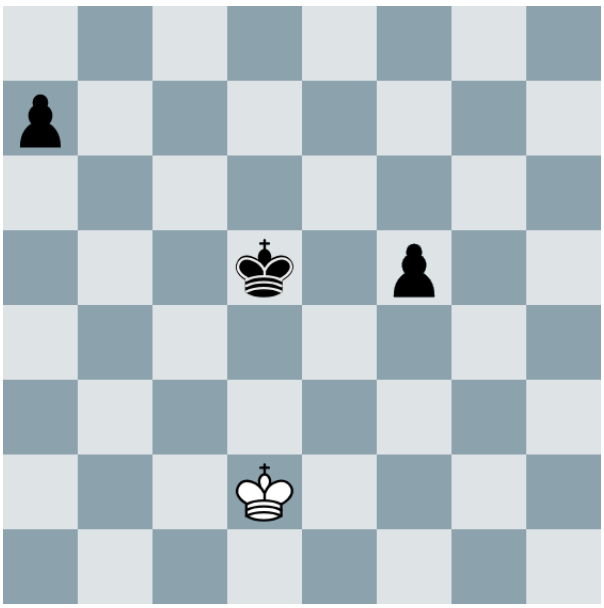


Figura 2.4: Posición 2

Si en lugar de simplemente haber aplicado el *minimax* hubiéramos aplicado la *quiescence search* ampliando la búsqueda después de la captura, hubiéramos descartado esa decisión y nos hubiéramos quedado con la posición 2.5, en la cual el jugador que tiene la posición ganadora es el que tiene las blancas.

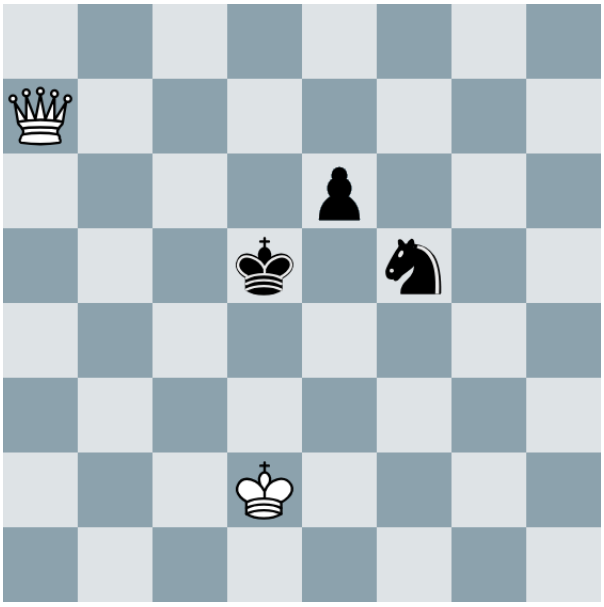


Figura 2.5: Posición 3

2.5. *Expectiminimax*

El algoritmo *expectiminimax* es una variante del algoritmo *minimax* que se utiliza en juegos cuyo resultado depende de una combinación de la habilidad del jugador y de eventos aleatorios como un lanzamiento de dados. Además de los nodos 'min' y 'max', en esta variante del *minimax* tenemos los nodos aleatorios, que toman como valor la esperanza matemática de un evento aleatorio, como un lanzamiento de dados.

En el *minimax* tradicional, los niveles del árbol se alternan entre 'max' y 'min' hasta que se alcanza el nivel de profundidad. En un árbol *expectiminimax*, los nodos aleatorios son intercalados con los nodos 'max' y 'min'. En vez de tomar el máximo o mínimo de los valores de sus hijos, los nodos de posibilidad toman una media ponderada, donde el peso de cada hijo es la probabilidad de que el hijo sea alcanzado.

La forma en la que se intercalan los nodos depende del juego. Por ejemplo, en un juego donde los jugadores lancen un dado al empezar su turno y en el que los turnos se intercalen (como el parchís) el orden de los nodos sería 'max', 'aleatorio', 'min' y después 'aleatorio'.

Algoritmo 2.13. El algoritmo en pseudocódigo sería el siguiente:

Algoritmo 6: *Expectiminimax*

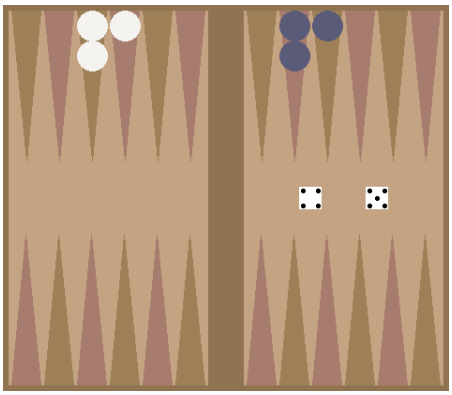
```
1  expectiminimax (nodo, profundidad, etiqueta)
2      if profundidad = 0 or nodo es final then
3          return heuristica(nodo, etiqueta)
4      else
5          if nodo es aleatorio then
6              valor ← 0;
7              foreach hijo de nodo do
8                  valor ← valor + (probabilidad(hijo) ·
                     expectiminimax(hijo, profundidad − 1, etiqueta));
9              return valor;
10         else
11             if etiqueta = "max" then
12                 valor ← −∞;
13                 foreach hijo de nodo do
14                     valor ←
                         máx{valor, expectiminimax(hijo, profundidad − 1, "min")};
15                 return valor;
16             else
17                 valor ← ∞;
18                 foreach hijo de nodo do
19                     valor ←
                         mín{valor, expectiminimax(hijo, profundidad − 1, "max")};
20                 return valor;
```

y para usarlo haremos una llamada inicial con $nodo = posicionActual$, $profundidad$ la que queramos y $etiqueta =$ "max" en nuestro turno.

Ejemplo 2.14 (*Expectiminimax* aplicado al *hypergammon*)**.** El *hypergammon* es una versión con menos fichas del *backgammon*. El objetivo del juego es llevar las fichas hasta el extremo contrario del tablero, para el jugador de fichas negras en sentido antihorario y para el jugador de fichas blancas al revés. Cada turno se lanzan dos dados y estos determinan cuantas casillas puedes mover dos fichas. No podemos colocar una ficha en una posición en la que haya más de dos fichas. Sin embargo, sí podemos colocar una ficha si solo hay una ficha del rival en una posición. De hecho, si lo hacemos, el rival tendrá que colocar su ficha en la barra central y no podrá mover otras fichas hasta que no saque esa ficha de la barra central.

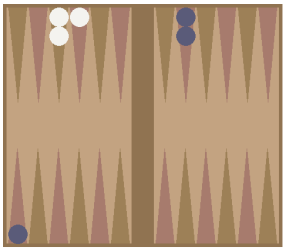
El número de combinaciones diferentes que pueden resultar de tirar los dados es $6+5+4+3+2+1=21$. Además, en el peor de los casos, sabiendo los dados podemos tener hasta $3^2=9$ jugadas legales si los dados son diferentes y hasta $3^4=81$ jugadas legales si los dados son iguales (si los dos dados tienen el mismo valor cuenta como si tuviéramos cuatro dados de ese valor).

Supongamos que nos encontramos en la siguiente posición, jugamos las fichas negras y los dados han sacado 4 y 5:

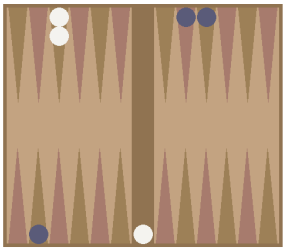


En este caso, por lo que mencionamos antes, el factor de ramificación es alto y llegar hasta tableros finales es demasiado costoso computacionalmente. Vamos a utilizar $profundidad = 3$ y a tomar la decisión que mejor valor devuelva. Para ello necesitamos definir una función heurística que nos valore posiciones no finales. La heurística va a ser la siguiente: para cada ficha vamos a contar el número de pasos que le quedan hasta el final del tablero, sumamos los de las fichas negras, sumamos los de las fichas blancas y calculamos la diferencia en función de para qué jugador estemos evaluando el tablero. Además, como en muchos casos el número de pasos va a coincidir para muchas de las jugadas posibles, vamos a sumar a la valoración anterior 2 puntos por cada ficha pasada (es decir, que no puede ser capturada) y 1 punto si tenemos 2 fichas en la misma posición (es decir, que bloquean su posición al rival).

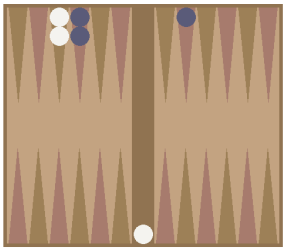
Los hijos de nuestra posición serían nodos aleatorios (ya que en ellos no se conoce el valor de los dados aún) y serían los siguientes:



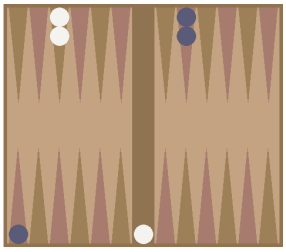
1



2



3



4

Lo siguiente, sería aplicar el *expectiminimax* con $profundidad = 2$ a cada uno de los cuatro hijos. Como estamos en un nodo aleatorio el resultado del siguiente nivel será la media de los resultados de aplicar el *expectiminimax* con $profundidad = 1$ a cada uno de los 21 hijos que tendrá cada hijo (en realidad lo que calculamos es el valor esperado, pero al ser todos los hijos equiprobables coincide con la media). El resultado sería el del árbol 2.6. Los valores debajo de cada hijo son los resultados. Entre ellos nos quedamos con el mayor (por ser el nodo inicial 'max') que es el hijo 4 con un valor de $-17{,}66$. Por tanto, el algoritmo devuelve $-17{,}66$ para el nodo inicial.

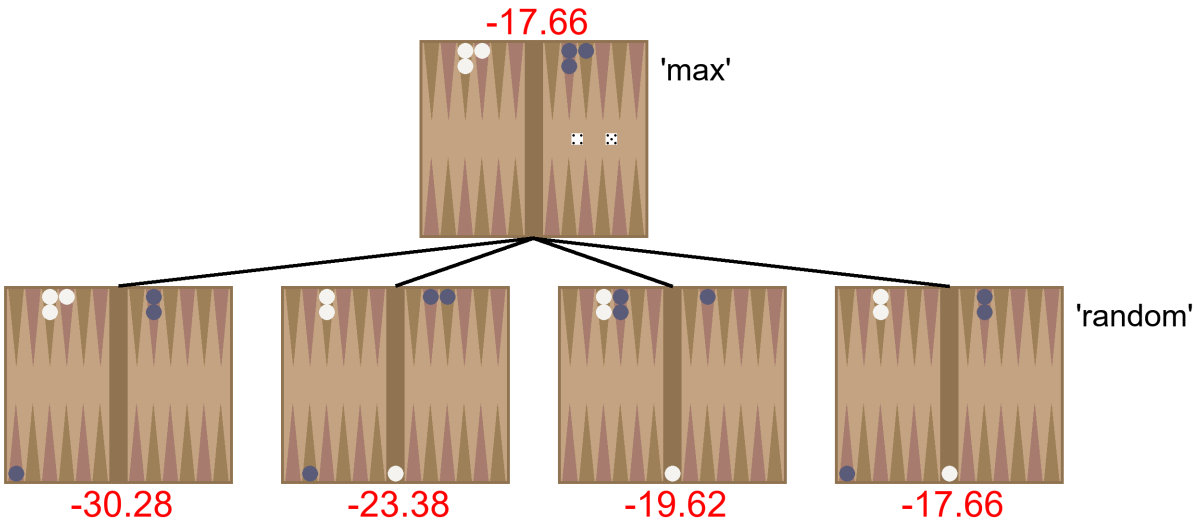


Figura 2.6: Árbol *expectiminimax*

Observación 2.15. El resultado del ejemplo anterior está influido por el hecho de que conocemos el valor de los dados. De no haberlo sabido el resultado hubiera sido $-24{,}93$. El peor resultado habría sido sacar 1 y 2 en los dados, con $-44{,}85$ y el mejor resultado habría sido sacar dos 4 en los dados, con $-10{,}19$.

Capítulo 3

Árbol de búsqueda de Monte Carlo (MCTS)

El algoritmo del árbol de búsqueda de Monte Carlo (o MCTS) toma las decisiones de una manera diferente a como lo hacen el *minimax* y sus variantes. La idea del algoritmo es simular muchas partidas desde la posición en que nos encontramos y a partir de los resultados de las simulaciones determinar cuál es la mejor jugada.

3.1. Funcionamiento del MCTS

El concepto principal del MCTS es la búsqueda. La búsqueda es el conjunto de recorridos a lo largo del árbol de juego. Un solo recorrido es un camino desde el nodo raíz (la posición actual del juego) hasta un nodo que no esté completamente expandido. Que un nodo no esté completamente expandido significa que al menos uno de sus hijos no ha sido explorado aún. Cuando un nodo que no está expandido es encontrado, uno de sus hijos que no ha sido explorado se elige para convertirse en la raíz de una simulación. Después, el resultado de la simulación se propaga hacia arriba hasta llegar al nodo raíz del árbol actual y se actualizan las estadísticas del nodo. Cuando el número de iteraciones fijado se alcanza, se elige un movimiento basándose en las estadísticas reunidas. Esta es la idea general del algoritmo, entremos ahora en los detalles.

3.1.1. Simulación

Una simulación es una secuencia de movimientos que empieza en el nodo actual (que representa una posición en un juego) y termina en un nodo terminal en el que se puede computar el resultado de la posición. Es decir, una simulación es una forma de evaluar los nodos a partir de correr de alguna manera una partida aleatoria empezando desde el nodo.

Durante la simulación, el siguiente movimiento se elige a partir de una función llamada *rollout policy* que dada una posición produce el siguiente movimiento. En la práctica, está diseñada para ser rápida para que las partidas puedan ser simuladas rápido y lo normal es tomar como *rollout policy* una función que elija aleatoriamente el movimiento.

Por tanto, una simulación en su forma más simple es una secuencia aleatoria de movimientos que empieza en una posición dada y la termina. Como la simulación llega siempre hasta una posición final, siempre llegamos hasta una evaluación, ya sea ganada, perdida o empate.

En el MCTS, las simulaciones siempre empiezan en un nodo que no ha sido explorado aún.

3.1.2. Expansión del árbol de juego

Dado un nodo y las reglas del juego, el resto del árbol de juego ya está determinado. Podemos recorrerlo sin necesidad de almacenarlo por completo en la memoria. Pero en su forma inicial está sin expandir. Al inicio de una partida nos encontramos en la raíz de un árbol de juego y el resto de nodos no han sido explorados aún. Cuando consideramos un movimiento nos imaginamos la posición en la que ese movimiento resultaría. Posiciones que ni siquiera se consideran visitar se quedan sin explorar y su potencial sin descubrir.

La misma distinción se aplica al árbol de juego del MCTS. Los nodos se consideran visitados o sin visitar. Un nodo se considera visitado si una simulación ha empezado en ese nodo, lo que quiere decir que ha sido evaluado al menos una vez. Si todos los hijos de un nodo han sido visitados se considera que este está completamente expandido, de otro modo no estaría completamente expandido y una expansión más profunda sería posible.

En la práctica, al inicio todos hijos del nodo raíz están sin visitar, uno se escoge y la primera simulación empieza.

Es importante tener en cuenta que los nodos elegidos por la función de *rollout policy* no se consideran visitados. Estos se consideran sin visitar aunque una simulación pase a través de ellos, solo el nodo donde la simulación empieza se considera visitado.

3.1.3. Retropropagación

Cuando una simulación para un nuevo nodo visitado termina, su resultado está listo para ser propagado hacia atrás hasta el nodo raíz del árbol actual.

La retropropagación es un recorrido hacia atrás desde el nodo donde empezó la simulación hasta el nodo raíz. El resultado de la simulación se sube hasta el nodo raíz y para todos los nodos en el camino de la retropropagación unas estadísticas son actualizadas. La retropropagación garantiza que las estadísticas de cada nodo reflejan los resultados de simulaciones empezadas en todos sus descendientes (al ser los resultados de las simulaciones propagados hasta el nodo raíz del árbol de juego).

3.1.4. Estadísticas de los nodos

La motivación de retropropagar los resultados de las simulaciones es actualizar $Q(v)$ (la recompensa total de las simulaciones) y $N(v)$ (el número total de visitas) para cada nodo v en el camino de la retropropagación (incluyendo el nodo donde empezó la simulación).

- $Q(v)$: la recompensa total de las simulaciones es un atributo de un nodo v y en su forma más simple es una suma de todos los resultados de simulaciones que han pasado por el mismo.
- $N(v)$: el número total de visitas es otro atributo de un nodo v que representa un contador de cuántas veces un nodo ha estado en el camino de la retropropagación (y por tanto, cuántas veces ha contribuido a la recompensa total de las simulaciones).

Estos dos valores se mantienen para todos los nodos visitados. Cuando un cierto número de simulaciones ha terminado, los nodos visitados tienen almacenada información indicando que tan explotados/explorados están.

En otras palabras, si miramos las estadísticas de un nodo cualquiera, estos dos valores reflejarán que tan prometedor el nodo es (recompensa total de las simulaciones) y que tan intensamente ha sido explorado (número total de visitas). Los nodos con una recompensa alta son buenos candidatos para seguir (explotación) pero los que tienen un número bajo de visitas también pueden ser interesantes (porque no han sido explorados bien).

3.1.5. Recorrido del árbol de juego

Al principio de la búsqueda, como no hemos empezado ninguna simulación aún, los nodos sin visitar son escogidos los primeros. Empezamos simulaciones únicas en cada uno de ellos, los resultados son retropropagados hasta el nodo raíz y después la raíz se considera completamente expandida.

Para elegir el siguiente nodo de nuestro camino en el que empezar la siguiente simulación desde un nodo v completamente expandido debemos considerar información de todos los hijos de v: v_1, v_2,...,v_k y la información del propio nodo v. El nodo v está completamente expandido por lo que ha tenido que ser visitado antes y acumula sus estadísticas: la recompensa total de las simulaciones ($Q(v)$) y el número total de visitas ($N(v)$), y lo mismo ocurre para sus hijos. Para elegir el siguiente nodo, además de estos valores necesitaremos una última pieza: la función UCT (o *upper confidence bound*)

UCT

El UCT es una función que nos permite elegir el siguiente nodo, entre varios nodos visitados, que vamos a atravesar. Es la función clave del MCTS y es la siguiente:

$$UCT(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log(N(v))}{N(v_i)}}$$

El nodo que maximice la función UCT será el siguiente a seguir durante el recorrido del árbol del MCTS. Veamos cómo funciona esta función:

La función está definida para un nodo hijo v_i de un nodo v. Es la suma de dos componentes, el primero es $\frac{Q(v_i)}{N(v_i)}$, también llamado componente de explotación, se puede entender como una tasa de victorias/derrotas, tenemos la recompensa total de las simulaciones entre el número total de visitas que estima el ratio de victorias de v_i. Pero no podemos quedarnos solo con esa componente ya que si no acabaríamos explorando solamente aquellos nodos que tengan una simulación victoriosa al inicio de la búsqueda. El segundo componente es $c\sqrt{\frac{\log(N(v))}{N(v_i)}}$, también llamado componente de exploración. Este componente favorece a los nodos que han sido poco explorados aún (es decir, aquellos con $N(v_i)$ bajo). El parámetro c sirve para controlar el balance entre explotación y exploración.

3.2. Pseudocódigo del MCTS

El algoritmo en pseudocódigo esta compuesto de varias funciones, la que nos devuelve la siguiente jugada es la función MCTS y sus parámetros de entrada son el número de simulaciones que queremos hacer y el nodo actual. El algoritmo en pseudocódigo (separado en partes) sería el siguiente:

Algoritmo 7: MCTS

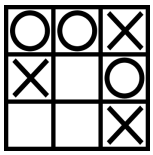
```
1 mcts (nodo, iteraciones)
2     while iteracion < iteraciones do
3         hoja ← atravesar(nodo);
4         resultado ← simulacion(hoja);
5         retropropagacion(hoja, resultado);
6     return mejorHijo(nodo);
```

```
1  atravesar (nodo)
2      if nodo no tiene hijos then
3          return nodo;
4      else
5          while completamenteExpandido(nodo) do
6              nodo ← mejorUCT(nodo);
7          return sinVisitar(nodo);
8  simulacion (nodo)
9      while nodo no es final do
10         nodo ← rolloutPolicy(nodo);
11     return resultado(nodo);
12 retropropagacion (nodo, resultado)
13     nodo.estadisticas ← actualizarEstadisticas(nodo, resultado);
14     if nodo no es raiz then
15         retropropagacion(nodo.padre, resultado);
```

donde la función $mejorUCT(nodo)$ escoge al hijo con mayor UCT de *nodo*, la función $completamenteExpandido(nodo)$ nos dice si *nodo* está completamente expandido o no, la función $sinVisitar(nodo)$ escoge un hijo sin visitar de *nodo*, la función $rolloutPolicy(nodo)$ escoge un hijo aleatorio de *nodo*, la función $actualizarEstadisticas(nodo, resultado)$ actualiza $N(nodo)$ y $Q(nodo)$ en función de *resultado* y por último, la función $mejorHijo(nodo)$ escoge al hijo con mayor número de visitas de *nodo*.

3.3. Ejemplo de aplicación del MCTS

Vamos a aplicar el algoritmo a la posición del *tic-tac-toe* que ya vimos para algunas variantes del *minimax*. Nos encontrábamos en la siguiente posición, era nuestro turno y jugábamos X:



Al aplicar el algoritmo con $iteraciones = 100$ las estadísticas para sus tres hijos son las que aparecen en la imagen 3.1. Como se puede ver, el hijo con mayor número de visitas es el 2, y por tanto, la decisión del MCTS en este caso coincide con la del *minimax* y sus variantes.

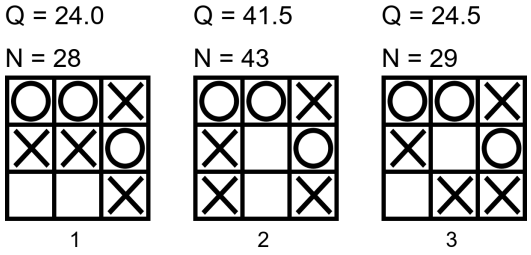


Figura 3.1: Estadísticas hijos

Capítulo 4

Aplicaciones en diferentes juegos

En esta sección veremos cual es el resultado de enfrentar los algoritmos descritos previamente a diferentes profundidades y en diferentes posiciones entre sí. También los enfrentaremos contra sí mismos a diferentes profundidades y a un algoritmo que toma las decisiones aleatoriamente para ver si realmente la toma de decisiones de los algoritmos es buena. Los juegos en los que vamos a enfrentar a los algoritmos son: el *connect* 4, el *reversi* y el *hypergammon*. Para los dos primeros juegos enfrentaremos a la poda alfa-beta contra el MCTS, para el último al ser un juego con factor aleatorio usaremos al *expectiminimax* en lugar de la poda alfa-beta.

4.1. *Connect* 4

El objetivo del *Connect* 4 es alinear cuatro fichas sobre un tablero formado por seis filas y siete columnas. Cada jugador dispone de 21 fichas de un color (en nuestro caso rojas y blancas). Por turnos, los jugadores deben introducir una ficha en la columna que prefieran (siempre que no esté completa) y ésta caerá a la posición más baja que todavía no esté ocupada. Gana la partida el primero que consiga alinear cuatro fichas consecutivas de un mismo color en horizontal, vertical o diagonal. Si todas las columnas están llenas pero nadie ha hecho una línea válida, hay empate.

La función heurística que vamos a utilizar para medir los tableros funciona de la siguiente manera: si queremos medir una posición no final para el jugador de fichas rojas, primero calculamos el número de líneas de 2 fichas rojas que se pueden convertir en líneas de 4 fichas rojas (es decir, no hay limitación ni por los bordes del tablero ni por las fichas del rival) y que no son sublíneas de una línea de 3, después hacemos lo mismo para las líneas de 3 fichas rojas. Repetimos el proceso anterior para las fichas blancas. Una vez sabemos esos 4 valores aplicamos la siguiente fórmula:

$$heuristica = peso2*(lineas2Rojas - lineas2Blancas) + peso3(lineas3Rojas - lineas3Blancas)$$

donde elegí como pesos $peso2 = \frac{1}{6}$ y $peso3 = \frac{5}{6}$.

Las posiciones en las que vamos a enfrentar los algoritmos son las de la imagen 4.1, donde la posición 1 es la posición inicial del juego. En las posiciones 1, 8 y 9 es el turno de las fichas rojas mientras que en el resto es el turno de las fichas blancas.

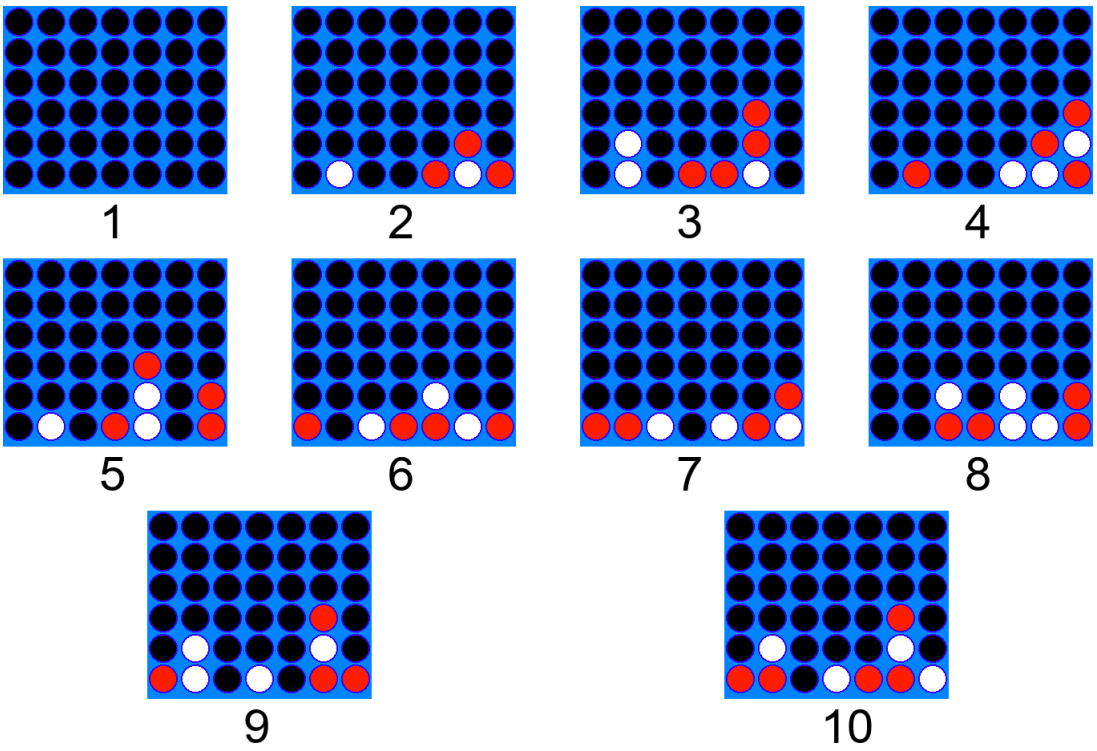


Figura 4.1: Posiciones *connect* 4

Para poder hacer un buen análisis de los resultados, los tiempos de decisión por jugada deben estar emparejados para los diferentes algoritmos. Los tiempos de decisión por jugada en segundos de la poda alfa-beta y del MCTS son los siguientes:

Profundidad	Tiempo (s)
3	0,02862707773844401
4	0,10944980382919312
5	0,47460474967956545
6	1,9352927684783936
7	6,156408965587616

Cuadro 4.1: Tiempos poda alfa-beta

Iteraciones	Tiempo (s)
8	0,024524075644356862
40	0,10873668844049628
170	0,473487483130561
590	1,9196439981460571
1750	6,1677347880143385
20000	53,97563099861145

Cuadro 4.2: Tiempos MCTS

Una partida promedio de *connect* 4 tiene 36 turnos, es decir, que en promedio una partida entre los algoritmos, dándoles el mismo tiempo por decisión, a su mayor profundidad debería

durar aproximadamente $36 * 6{,}16 = 221{,}76s$. Esta duración de 3 minutos y 41 segundos nos hace pensar que la profundidad de búsqueda podría ser extendida, pero en la práctica no es viable aumentar la profundidad de la poda alfa-beta sin que colapse la memoria. En el caso del MCTS la memoria sí nos permite hacer un mayor número de iteraciones pero si buscásemos a más profundidad los tiempos de decisión no estarían emparejados. En cualquier caso, el hecho de no colapsar la memoria tan rápido supone una clara ventaja para el algoritmo MCTS si podemos disponer de más tiempo de ejecución. Analizar cuáles serían los resultados dejándolo a una profundidad descompensada es interesante, por ello vamos a incluir el caso en el que $iteraciones = 20000$.

Los resultados de enfrentar a los algoritmos en la posiciones anteriores han sido los siguientes (en formato fichas blancas - fichas rojas):

		Fichas rojas					
		Poda 3	Poda 4	Poda 5	Poda 6	Poda 7	Random
Fichas blancas	Poda 3	4 - 6	4 - 6	2 - 8	2 - 8	0 - 10	10 - 0
	Poda 4	7 - 3	5 - 5	5.5 - 4.5	2 - 8	1 - 9	10 - 0
	Poda 5	7 - 3	5.5 - 4.5	5.5 - 4.5	5.5 - 4.5	3 - 7	10 - 0
	Poda 6	9.5 - 0.5	6 - 4	8 - 2	4 - 6	3 - 7	10 - 0
	Poda 7	8.5 - 1.5	5.5 - 4.5	6.5 - 3.5	4.5 - 5.5	7 - 3	10 - 0
	Random	0 - 10	0 - 10	0 - 10	0 - 10	0 - 10	3 - 7

Cuadro 4.3: Resultados poda alfa-beta

		Fichas rojas						
		MCTS 8	MCTS 40	MCTS 170	MCTS 590	MCTS 1750	MCTS 20000	Random
Fichas blancas	MCTS 8	5 - 5	2 - 8	1 - 9	0 - 10	0 - 10	0 - 10	7 - 3
	MCTS 40	8 - 2	7 - 3	3 - 7	2 - 8	2 - 8	1 - 9	10 - 0
	MCTS 170	10 - 0	9 - 1	4 - 6	5 - 5	5 - 5	4 - 6	10 - 0
	MCTS 590	10 - 0	6 - 4	8 - 2	5 - 5	6 - 4	6 - 4	10 - 0
	MCTS 1750	7 - 3	7 - 3	6 - 4	4 - 6	5 - 5	4 - 6	10 - 0
	MCTS 20000	9 - 1	7 - 3	5 - 5	5 - 5	4 - 6	5 - 5	10 - 0
	Random	2 - 8	2 - 8	0 - 10	0 - 10	0 - 10	0 - 10	3 - 7

Cuadro 4.4: Resultados MCTS

		Fichas rojas				
		Poda 3	Poda 4	Poda 5	Poda 6	Poda 7
Fichas blancas	MCTS 8	0 - 10	0 - 10	0 - 10	0 - 10	0 - 10
	MCTS 40	1 - 9	1 - 9	0 - 10	0 - 10	1 - 9
	MCTS 170	3 - 7	2 - 8	2 - 8	2 - 8	1 - 9
	MCTS 590	3 - 7	2.5 - 7.5	1.5 - 8.5	1.5 - 8.5	1 - 9
	MCTS 1750	3 - 7	2 - 8	1 - 9	0 - 10	0 - 10
	MCTS 20000	4 - 6	2 - 8	1 - 9	0 - 10	0 - 10

Cuadro 4.5: Resultados MCTS - Poda alfa-beta

		Fichas rojas					
		MCTS 8	MCTS 40	MCTS 170	MCTS 590	MCTS 1750	MCTS 20000
Fichas blancas	Poda 3	9 - 1	8 - 2	5 - 5	5 - 5	6 - 4	6 - 4
	Poda 4	10 - 0	9 - 1	8 - 2	7 - 3	8 - 2	9 - 1
	Poda 5	10 - 0	9 - 1	8 - 2	9 - 1	10 - 0	10 - 0
	Poda 6	10 - 0	10 - 0	7 - 3	10 - 0	9 - 1	10 - 0
	Poda 7	10 - 0	10 - 0	9 - 1	9 - 1	9 - 1	10 - 0

Cuadro 4.6: Resultados Poda alfa-beta - MCTS

Los dos algoritmos muestran resultados casi perfectos contra el algoritmo aleatorio, por lo que podemos concluir que están funcionando bien. Los resultados del MCTS contra sí mismo al ir aumentando los tiempos no parecen tener una gran mejora, a partir de $iteraciones = 170$ todas las iteraciones funcionan de manera similar, mientras que para la poda alfa-beta los resultados mejoran ligeramente a mayor profundidad. Por último, para los enfrentamientos de un algoritmo contra otro, el claro ganador es la poda alfa-beta, gana prácticamente todas las combinaciones de profundidades e iteraciones y muchas veces con una gran diferencia (incluso para el número máximo de iteraciones del MCTS).

Los enfrentamientos de algoritmos contra si mismos a misma profundidad son bastante igualados en general, esto nos hace pensar que las posiciones iniciales están equilibradas para los diferentes colores de fichas.

4.2. *Reversi*

El reversi es un juego entre dos personas, que comparten 64 fichas iguales, de caras distintas, que se van colocando por turnos en un tablero de ocho filas y ocho columnas. Las caras de las fichas se distinguen por su color (en nuestro caso blancas y negras) y cada jugador tiene asignado uno de esos colores. Por turnos, los jugadores colocan una ficha en el tablero (a menos que no tengan ninguna jugada legal) de forma que flanquee una o varias fichas del color contrario y se voltean esas fichas para que pasen a mostrar el color propio. Gana el jugador que tenga más fichas sobre el tablero al finalizar la partida.

La función heurística que vamos a utilizar en este caso es simple, vamos a contar las fichas de ambos jugadores en el tablero y a calcular la diferencia (en función de para quien calculemos la heurística se hará la diferencia de una manera u otra).

Las posiciones en las que vamos a enfrentar los algoritmos son las de la imagen 4.2, donde la posición 1 es la posición inicial del juego. En todas ellas es el turno de las fichas negras.

Al igual que con el *connect* 4 los tiempos por decisión de cada algoritmo están emparejados para facilitar el análisis de los resultados. Los tiempos en este caso son los de las tablas 4.7 y 4.8.

Profundidad	Tiempo (s)
2	0,030983233451843263
3	0,17031026970256458
4	1,1829345533924718
5	2,423319237572806
6	15,855559428532919

Cuadro 4.7: Tiempos poda alfa-beta

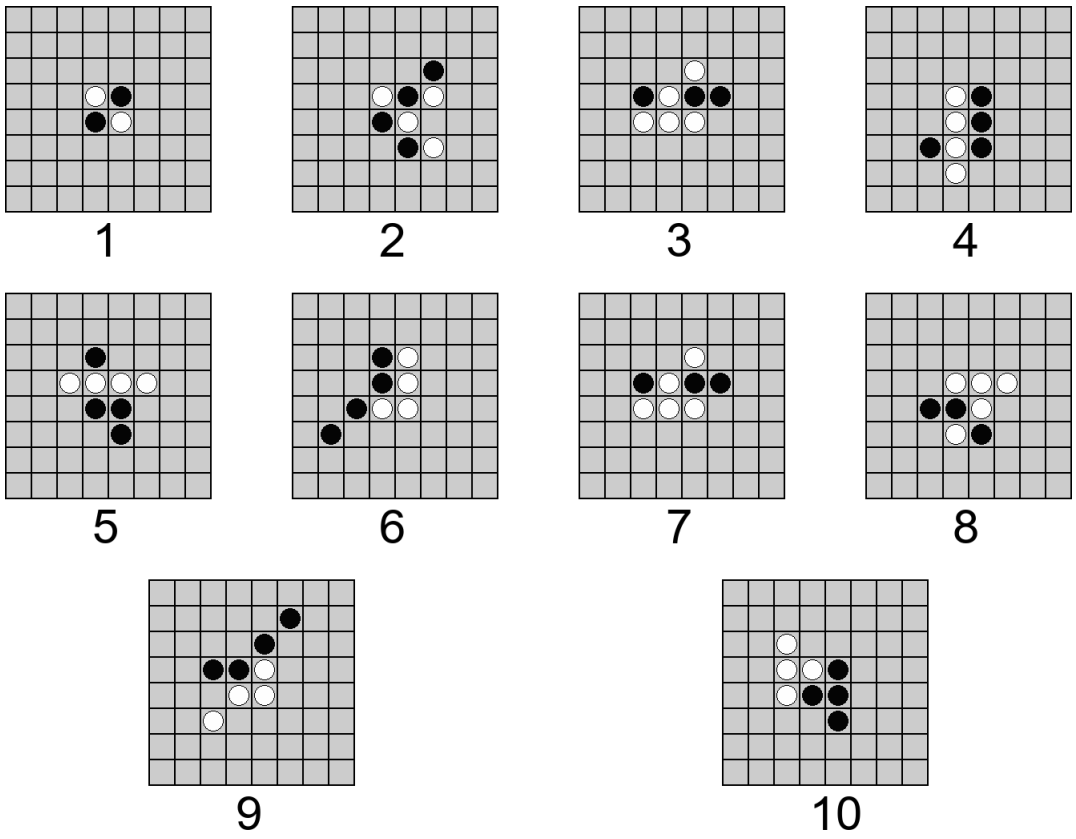


Figura 4.2: Posiciones *reversi*

Iteraciones	Tiempo (s)
1	0,0805530865987142
3	0,17514230410257975
13	1,2137874480216735
26	2,3641257593708653
170	15,709569854121055
1200	54,23565186659495

Cuadro 4.8: Tiempos MCTS

Como con el *connect* 4, el número de iteraciones del MCTS podría ser extendido, pero si extendiésemos la profundidad de búsqueda de la poda alfa-beta colapsaría la memoria. Además, el factor de ramificación es más alto en este juego que en el *connect* 4, es por ello que para la misma profundidad de búsqueda los tiempos por decisión son más altos en este caso. Como el MCTS tiene la ventaja de no colapsar la memoria tan rápidamente vamos a dejarle un número de iteraciones extra (1200 iteraciones) para ver qué pasaría si dispusiésemos de tiempos de ejecución más grandes por decisión.

Los resultados de enfrentar a los algoritmos en la posiciones anteriores han sido los siguientes (en formato fichas blancas - fichas negras):

		Fichas negras					
		Poda 2	Poda 3	Poda 4	Poda 5	Poda 6	Random
Fichas blancas	Poda 2	6 - 4	7 - 3	2 - 8	3 - 7	2 - 8	7 - 3
	Poda 3	9 - 1	5 - 5	5 - 5	2 - 8	1 - 9	8 - 2
	Poda 4	6.5 - 3.5	5 - 5	8 - 2	2 - 8	3 - 7	9 - 1
	Poda 5	9 - 1	9 - 1	7 - 3	7 - 3	5 - 5	9 - 1
	Poda 6	8 - 2	10 - 0	9 - 1	6 - 4	7.5 - 2.5	10 - 0
	Random	2 - 8	2 - 8	1 - 9	3 - 7	0 - 10	4.5 - 5.5

Cuadro 4.9: Resultados poda alfa-beta

		Fichas negras						
		MCTS 1	MCTS 3	MCTS 13	MCTS 26	MCTS 170	MCTS 1200	Random
Fichas blancas	MCTS 1	6 - 4	8 - 2	0 - 10	1 - 9	0 - 10	0 - 10	5 - 5
	MCTS 3	4 - 6	3 - 7	5 - 5	0 - 10	0 - 10	0 - 10	4 - 6
	MCTS 13	7 - 3	8 - 2	7 - 3	3 - 7	2 - 8	0 - 10	8 - 2
	MCTS 26	10 - 0	8 - 2	8 - 2	3 - 7	0 - 10	2 - 8	9 - 1
	MCTS 170	10 - 0	10 - 0	9 - 1	8 - 2	6 - 4	1 - 9	10 - 0
	MCTS 1200	10 - 0	10 - 0	10 - 0	7 - 3	9 - 1	4 - 6	10 - 0
	Random	2 - 8	6 - 4	2 - 8	2 - 8	0 - 10	0 - 10	4.5 - 5.5

Cuadro 4.10: Resultados MCTS

		Fichas negras				
		Poda 2	Poda 3	Poda 4	Poda 5	Poda 6
Fichas blancas	MCTS 1	2 - 8	2.5 - 7.5	3 - 7	1 - 9	0 - 10
	MCTS 3	3 - 7	3.5 - 6.5	0 - 10	2 - 8	1 - 9
	MCTS 13	7 - 3	3.5 - 6.5	4 - 6	2 - 8	4 - 6
	MCTS 26	10 - 0	6 - 4	6 - 4	8 - 2	7 - 3
	MCTS 170	8 - 2	10 - 0	6 - 4	8 - 2	7 - 3
	MCTS 1200	10 - 0	10 - 0	9 - 1	10 - 0	9 - 1

Cuadro 4.11: Resultados MCTS - Poda alfa-beta

		Fichas negras					
		MCTS 1	MCTS 3	MCTS 13	MCTS 26	MCTS 170	MCTS 1200
Fichas blancas	Poda 2	6 - 4	7.5 - 2.5	3 - 7	0 - 10	0 - 10	0 - 10
	Poda 3	6.5 - 3.5	10 - 0	6 - 4	3 - 7	1 - 9	0 - 10
	Poda 4	9 - 1	8 - 2	3 - 7	3 - 7	0 - 10	0 - 10
	Poda 5	10 - 0	9.5 - 0.5	9 - 1	5.5 - 4.5	2 - 8	0 - 10
	Poda 6	7 - 3	10 - 0	6 - 4	7 - 3	1 - 9	0 - 10

Cuadro 4.12: Resultados Poda alfa-beta - MCTS

Los dos algoritmos muestran resultados buenos contra el algoritmo aleatorio, salvo las versiones con menos iteraciones (1 y 3) del MCTS, esto se debe a que el número de partidas simuladas es muy bajo. Los resultados del MCTS contra sí mismo al ir aumentando los

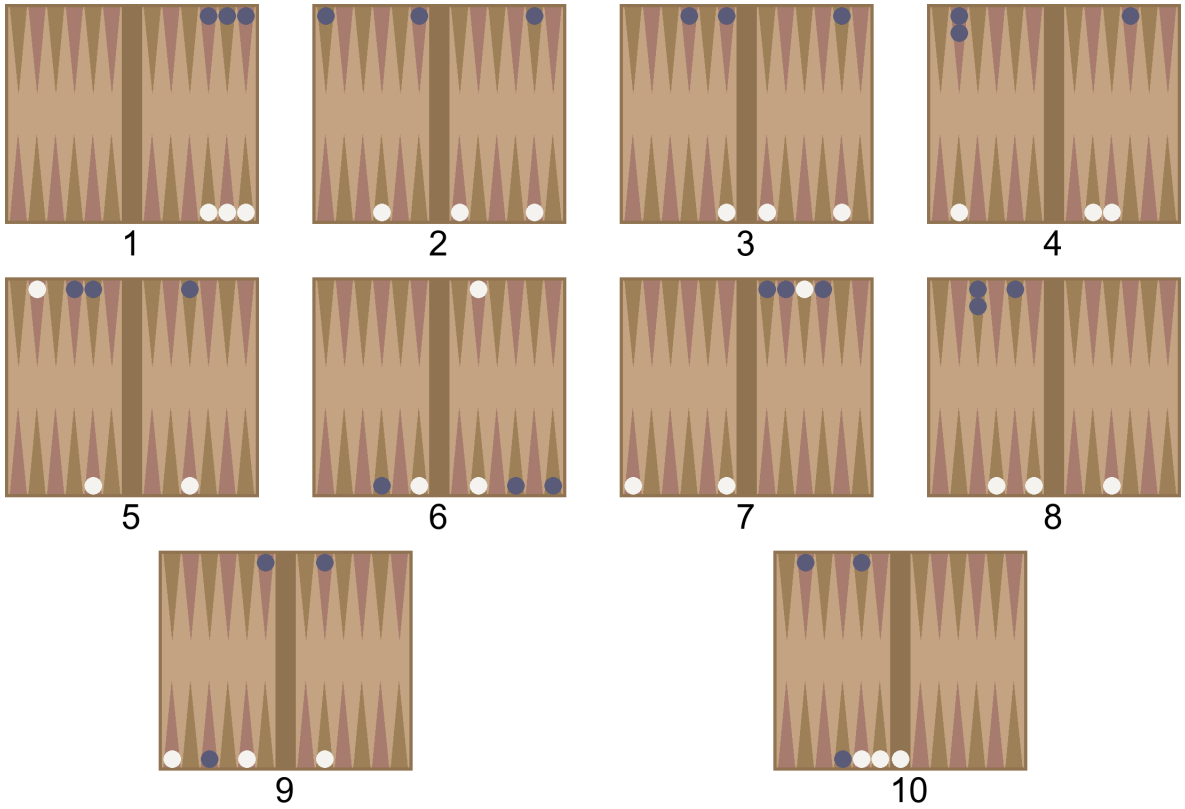
tiempos mejoran claramente (salvo para los casos 1 y 3 de nuevo), mientras que para la poda alfa-beta aunque los resultados también mejoran a mayor profundidad, los resultados son más igualados. Por último, para los enfrentamientos de un algoritmo contra otro, la poda alfa-beta gana los casos a menor profundidad mientras que el MCTS es mejor en los casos con más iteraciones. En concreto, la profundidad extra que nos permite el MCTS tiene resultados casi perfectos contra la poda alfa-beta

En los enfrentamientos de la poda alfa-beta contra sí misma a misma profundidad, hay una tendencia a que las fichas blancas ganen más. Si sumamos los resultados de las 50 partidas, el resultado final sería 33.5 - 16.5, es decir que las fichas blancas ganan más del doble que las fichas negras. Esta tendencia no aparece en el MCTS, donde la suma de resultados es de 29 - 31.

4.3. *Hypergammon*

Como ya vimos en el ejemplo en el que aplicamos el algoritmo *expectiminimax* al *hypergammon*, este juego tiene un factor aleatorio. Esto hace que en lugar de la poda alfa-beta clásica tengamos que usar el *expectiminimax* en las partidas. La heurística que vamos a usar en este caso es la descrita en el ejemplo del *expectiminimax*.

Las posiciones en las que vamos a enfrentar los algoritmos son las siguientes:



donde la posición 1 es la posición inicial del juego. En las posiciones 8, 9 y 10 es el turno de las fichas blancas mientras que en el resto es el turno de las fichas negras.

En este caso la forma de emparejar los tiempos será algo diferente. Para cada nodo en el que se tiran los dados como ya vimos el número de hijos es de 21. Esto hace que el factor de ramificación sea demasiado alto y que los problemas de memoria aparezcan a menor profundidad de búsqueda. Hay dos formas de lidiar con este problema: buscar a

baja profundidad o limitar la búsqueda de manera que no evaluemos el árbol para algunos valores de los dados. En la práctica, limitar la búsqueda permitió llegar a un nivel más de profundidad con el algoritmo. Otra de las diferencias con los juegos anteriores es que al usar menor profundidad en la variante del *minimax*, los tiempos no pueden ser emparejados con tiempos del MCTS (es decir, para el menor número de iteraciones del MCTS el tiempo ya supera a la mayor profundidad del *expectiminimax* la cual no puede ser aumentada). Teniendo en cuenta esto, los tiempos en este caso son los siguientes:

Profundidad	Tiempo (s)
1	0,0002368821038140191
2	0,7719471262704308
3	0,9240370213813425

Cuadro 4.13: Tiempos *expectiminimax*

Profundidad	Tiempo (s)
1	0,00015363326439490684
2	0,7447383063180106
3	0,8291703462600708
4	60,8937874480130561

Cuadro 4.14: Tiempos *expectiminimax* limitado

Iteraciones	Tiempo (s)
60	59,78140670602971

Cuadro 4.15: Tiempos MCTS

Como se puede ver los tiempos del *expectiminimax* limitado son ligeramente inferiores a los del *expectiminimax*, pero al llegar a $profundidad = 4$ el tiempo por decisión aumenta drásticamente. El número de iteraciones del MCTS está ajustado para que se pueda comparar sus decisiones con las del *expectiminimax* limitado con $profundidad = 4$. De hecho, para $iteraciones = 60$ el MCTS está cerca del límite en memoria por lo que no se puede extender más en este caso.

Los resultados de enfrentar a los algoritmos en la posiciones anteriores han sido los siguientes (en formato fichas blancas - fichas negras):

		Fichas negras				
		Em lim 1	Em lim 2	Em lim 3	Em lim 4	Random
Fichas blancas	Em lim 1	3 - 7	6 - 4	2 - 8	4 - 6	8 - 2
	Em lim 2	3 - 7	5 - 5	4 - 6	3 - 7	8 - 2
	Em lim 3	6 - 4	5 - 5	5 - 5	2 - 8	8 - 2
	Em lim 4	6 - 4	5 - 5	6 - 4	4 - 6	8 - 2
	Random	2 - 8	1 - 9	1 - 9	1 - 9	4 - 6

Cuadro 4.16: Resultados *expectiminimax* limitado

		Fichas negras				
		Em 1	Em 2	Em 3	MCTS 60	Random
Fichas blancas	Em 1	2 - 8	6 - 4	2 - 8	4 - 6	9 - 1
	Em 2	8 - 2	3 - 7	3 - 7	6 - 4	7 - 3
	Em 3	4 - 6	4 - 6	7 - 3	8 - 2	9 - 1
	MCTS 60	3 - 7	2 - 8	2 - 8	4 - 6	7 - 3
	Random	0 - 10	1 - 9	1 - 9	1 - 9	4 - 6

Cuadro 4.17: Resultados *expectiminimax* y MCTS

		Fichas negras			
		Em lim 1	Em lim 2	Em lim 3	Em lim 4
Fichas blancas	Em 1	6 - 4	8 - 2	2 - 8	3 - 7
	Em 2	5 - 5	6 - 4	3 - 7	3 - 7
	Em 3	6 - 4	6 - 4	4 - 6	8 - 2
	MCTS 60	4 - 6	2 - 8	1 - 9	2 - 8

Cuadro 4.18: Resultados *Expectiminimax* y MCTS - *Expectiminimax* limitado

		Fichas negras			
		Em 1	Em 2	Em 3	MCTS 60
Fichas blancas	Em lim 1	5 - 5	3 - 7	3 - 7	3 - 7
	Em lim 2	4 - 6	4 - 6	3 - 7	5 - 5
	Em lim 3	5 - 5	5 - 5	2 - 8	6 - 4
	Em lim 4	5 - 5	6 - 4	4 - 6	7 - 3

Cuadro 4.19: Resultados *Expectiminimax* limitado - *Expectiminimax* y MCTS

Los tres algoritmos muestran resultados muy buenos contra el algoritmo aleatorio. Los resultados del *expectiminimax* limitado contra sí mismo son ligeramente mejores a mayor profundidad y pasa lo mismo para el *expectiminimax*. Por último, para el enfrentamiento del MCTS contra los otros dos algoritmos, el MCTS a pesar de la ventaja de tiempo muestra malos resultados contra las dos versiones del *expectiminimax*. Por último, en el enfrentamiento entre los dos *expectiminimax*, en igualdad de condiciones y en la mayor profundidad de ambos gana la versión sin limitar, pero los resultados de la versión limitada son buenos.

En este caso, los resultados para un algoritmo contra sí mismo a la misma profundidad, muestran un poco de ventaja para las fichas negras.

Capítulo 5

Conclusiones

Tanto los algoritmos variantes del *minimax* como el MCTS muestran buenos resultados al ser enfrentados al algoritmo aleatorio en los diversos juegos. Además, a más profundidad mejores son estos resultados. A partir de esto se puede concluir que ambos algoritmos proporcionan una buena toma de decisiones y que la profundidad los mejora.

Sin embargo, las variantes del *minimax* parecen, en general, una mejor forma de abordar el problema (por lo menos si no tenemos en cuenta los problemas de memoria). Da la impresión de que con una heurística apropiada estos algoritmos siempre van a estar por encima del MCTS. En el caso del *connect* 4, el juego es muy intuitivo y la heurística dada parece reflejar bien como se deberían medir las posiciones, y en ese caso los resultados son muy favorables para la poda alfa-beta. El *reversi* es un juego más complejo, ya que, a pesar de tener mucha ventaja en número de fichas en un momento de la partida, las cosas pueden cambiar repentinamente. Dar una heurística que sea capaz de detectar qué tableros son susceptibles de estos cambios repentinos de la evaluación es complicado. Y mirando a baja profundidad, solo contar el número de fichas no parece ser suficiente. Por ello los resultados en este juego son más igualados. En el *hypergammon* dar una buena heurística también es complicado, pero con una heurística simple como la usada se ven buenos resultados para el *expectiminimax*.

También, se debe tener en cuenta que el *minimax* y sus variantes son algoritmos más intuitivos y que abordan el problema de una forma más humana. En especial, la *quiescence search* es el algoritmo que tiene una forma de funcionar más cercana al pensamiento de un humano y por ello me parece la más interesante. A cada posición del tablero un buen jugador es capaz de ver qué jugadas son sustanciales y cuáles no, y en función de eso profundiza en el subárbol de una jugada o no. El algoritmo funciona de la misma manera.

Hay que tener en cuenta también que los problemas de memoria del *minimax* estan ahí. En juegos con un factor de ramificación más alto, como el *go* (que tiene un factor de ramificación de 250), abordar el problema con estos algoritmos es inviable. Es por ello que algoritmos como el de AlphaGo Lee, que juega al *go*, utilizan versiones del MCTS mejoradas.

En conclusión, los dos enfoques (*minimax* y MCTS) son razonables y ambos tienen contextos en los que son mejores que el otro. Uno con un enfoque más intuitivo y el otro con un enfoque mejor para situaciones en las que no se pueda explorar tan exhaustivamente.

Bibliografía

[1] Carrillo, M. código tfg. https://github.com/MariocarrilloGH/TFG, 2023.

[2] Czarnogorski, K. Monte carlo tree search – beginners guide. https://int8.io/monte-carlo-tree-search-beginners-guide/#Policy_network_training_in_Alpha_Go_and_Alpha_Zero, Mar. 2018.

[3] Mayefsky, E., Anene, F., and Sirota, M. Strategies and tactics for intelligent search. https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/minimax.html, 2003.

[4] Mayefsky, E., Anene, F., and Sirota, M. Strategies and tactics for intelligent search. https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/alphabeta.html, 2003.

[5] Wikipedia. Conecta 4 — wikipedia, la enciclopedia libre. https://es.wikipedia.org/w/index.php?title=Conecta_4&oldid=136349789, 2021.

[6] Wikipedia. Backgammon — wikipedia, la enciclopedia libre. https://es.wikipedia.org/w/index.php?title=Backgammon&oldid=151557867, 2023.

[7] Wikipedia. Reversi — wikipedia, la enciclopedia libre. https://es.wikipedia.org/w/index.php?title=Reversi&oldid=151272752, 2023.

[8] Wikipedia contributors. Branching factor — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Branching_factor&oldid=1020914491, 2021.

[9] Wikipedia contributors. Principal variation search — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Principal_variation_search&oldid=1032100568, 2021.

[10] Wikipedia contributors. Quiescence search — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Quiescence_search&oldid=1122821044, 2022.

[11] Wikipedia contributors. Alpha–beta pruning — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=1157654948, 2023.

[12] Wikipedia contributors. Expectiminimax — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Expectiminimax&oldid=1154986117, 2023.

[13] Wikipedia contributors. Game complexity — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Game_complexity&oldid=1160857044, 2023.

[14] Wikipedia contributors. Minimax — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Minimax&oldid=1160191816, 2023.

[15] Wikipedia contributors. Tic-tac-toe — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Tic-tac-toe&oldid=1159091894, 2023.