# Minimax

June 26, 2017

**Abstract**

Minimax es una regla de decisión usada en la teoría de la decisión, la teoría de juegos, las estadísticas y la filosofía para minimizar la posible pérdida para un escenario del peor caso(pérdida máxima). Originalmente formulado para la teoría de juego de dos jugadores de suma cero, que abarca tanto los casos en que los jugadores toman movimientos alternativos como aquellos en los que realizan movimientos simultáneos, también se ha extendido a juegos más complejos ya la toma de decisiones generales en presencia de incertidumbre. En el presente informe se realizará la presentación del algoritmo donde se de la ganancia optima utilizando sistemas distribuidos y concurrentes mediante la estructura de un cliente, servidor y trabajador.

keywords: Teoría de juego, Zero-sum game, Teoría de la decisión

## 1 Introducción

Existe un montón de aplicaciones en el campo de la IA, pero los juegos son los más interesantes y populares en los cuales han sido utilizado. Estos se aplica en juegos de 2 jugadores como Tic Tac Toe, damas, ajedrez, go, etc. Todos estos juegos tienen al menos una cosa en común, son **juegos de lógica**. Esto significa que pueden ser descritos por un conjunto de reglas y premisas. Con ellos, es posible saber desde un punto determinado en el juego, cuáles son los próximos movimientos disponibles. Otra característica, son **juegos de información completa**. Cada jugador sabe todo acerca de los posibles movimientos del adversario, por ultimo son formulados dentro de la teoria de juegos de 2 jugadores de suma cero, el cual posee las siguientes características:

- Son los más simples.
- 2 oponentes compitiendo que juegan por turnos alternos.
- Entorno:
  - Determinístico.
  - Totalmente observable.
- Lo que gana uno, lo pierde el otro.

Otro concepto también importante son los arboles de búsqueda donde estos juegos se desarrollan, pues estos tienen características a tomar al momento de la elección del movimiento mas optimo, las cuales son:

- Alto factor de ramificación, mucha profundidad, se exige el óptimo.
- Tiempo limitado, falta de información, elementos de azar.
  - 3 en Raya: 9! = 362.880 nodos.
  - Ajedrez: $10^{40}$ nodos.
- Necesario: heurística, utilidad, poda

En este sentido se tiene dos jugadores involucrados, **MAX y MIN**. Conociendo cada unas de las posibles jugadas y su score respectivo se genera un árbol de búsqueda, DFS, comenzando con la posición actual del juego hasta la posición final del juego. A continuación, se evalúa la posición final del juego desde el punto de vista de MAX. Posteriormente, los valores de los nodos internos del árbol se van rellenando de abajo hacia arriba con los valores evaluados. Los nodos que pertenecen al jugador MAX reciben el valor máximo de sus hijos. Los nodos para el jugador MIN seleccionarán el valor mínimo de sus hijos. Los valores que se conocen inicialmente estarán ubicados en los nodos hojas y mediante el algoritmo llegar hasta el nodo raiz.

Entonces, los valores representan lo optimo que es un movimiento del juego ya sea para MAX o MIN. Así que el jugador MAX intentará seleccionar el movimiento con el valor más alto al final. Pero el jugador MIN tratará de seleccionar los movimientos que son mejores para él, minimizando así el resultado de MAX.

# 2 Optimización

Utilizando solo juegos simples se puede obtener el árbol de búsqueda completo en poco tiempo. Pero para la mayoría de los juegos esto no es posible, el universo probablemente desaparecería primero. Así que hay algunas optimizaciones para agregar al algoritmo.

Debemos saber que al optimizar se ignorará parte de la información sobre los eventos del juego con probabilidades y atajos. En lugar de conocer el camino completo que conduce a la victoria, las decisiones se toman con el camino que podría conducir a la victoria. Si la optimización no está bien escogida, o está mal aplicada, entonces podríamos terminar con una IA muda. Y habría sido mejor usar movimientos aleatorios.

- Una optimización básica es limitar la profundidad del árbol de búsqueda. Esto nos ayuda debido a que si tenemos en frente a un problema complejo entonces generar el árbol completo podría tomar mucho tiempo y además poder recorrerlo. Si un juego tiene un factor de ramificación de 3, esto significaría que cada nodo 3 tiene hijos en el árbol.

| Depth | Node Count |
|---|---|
| 0 | 1 |
| 1 | 3 |
| 2 | 9 |
| 3 | 27 |
| ... | ... |
| n | $3^n$ |

Figure 1: Juego con un factor de ramificación 3

La secuencia muestra que a la profundidad n el árbol tendrá $3^n$ nodos. Para conocer el número total de nodos generados, necesitamos sumar la cantidad de nodos en cada nivel. Así que el número total de nodos para un árbol con profundidad n es $\sum_{n=0}^{n} 3^n$. Para muchos juegos, como el ajedrez que tienen un factor de ramificación muy grande, esto significa que el árbol podría no ser bueno para su uso en la memoria.

Incluso si este lo hiciera, tomaría mucho tiempo para generar. Supongamos que cada nodo tome 1s para ser analizado, eso significaría que para la optimización anterior, cada árbol de búsqueda tomaría $\sum_{n=0}^{n} 3^n$. Para un árbol de búsqueda con profundidad 5, eso significaría $1 + 3 + 9 + 27 + 81 + 243 = 364 * 1 = 364s => 6minutos!$. Esto es demasiado largo para un juego.

- La segunda optimización es utilizar una función que evalúa la posición actual del juego desde el punto de vista de algún jugador. Esto lo hace dando un valor al estado actual del juego, como contar el número de piezas en el tablero, por ejemplo, el número de movimientos dejados al final del juego, o cualquier otra cosa que podamos usar para dar un valor a la posición del juego.

En lugar de evaluar la posición actual del juego, la función podría calcular cómo la posición actual del juego podría ayudar a terminar el juego. O en otras palabras, la probabilidad de que con la posición actual del juego, podamos ganar el juego. En este caso, la función se conoce como **función de estimación**.

Esta función tendrá que tener en cuenta algunas heurísticas. Heurística son los conocimientos que tenemos sobre el juego, y ayudar a generar mejores funciones de evaluación. Por ejemplo, en las DAMAS, las piezas en las esquinas y en las posiciones laterales no se pueden comer. Así podemos crear una función de evaluación que da valores más altos a las piezas que se encuentran en esas posiciones del tablero, dando así mayores resultados para los movimientos del juego que coloquen piezas en esas posiciones.

Una de las razones por las que la función de evaluación debe ser capaz de evaluar posiciones de juego para ambos jugadores es que no sabe a qué jugador pertenece la profundidad del límite. Sin embargo, se

pueden evitar dos funciones si el juego es simétrico. Esto significa que la pérdida de un jugador es igual a las ganancias del otro. Estos juegos también se conocen como juegos ZERO-SUM. Para estos juegos una función de evaluación es suficiente, uno de los jugadores sólo tiene que negar el retorno de la función.

Aun así el algoritmo tiene algunos defectos, uno de los defectos es que si el juego es demasiado complejo la respuesta siempre tomará demasiado tiempo incluso con un límite de profundidad. Una solución es limitar el tiempo de búsqueda. Es decir, si el tiempo se agota se elige el mejor movimiento encontrado hasta el momento.

Un gran defecto es el problema horizonte limitado pues una posición de juego que parece ser muy buena puede resultar muy mala. Esto sucede porque el algoritmo no fue capaz de ver que unos pocos movimientos de juego por delante el adversario será capaz de hacer un movimiento que le traerá un gran resultado. El algoritmo perdió ese movimiento fatal porque estaba cegado por el límite de profundidad.

# 3 Aceleración del algoritmo

Vemos anteriormente que el problema principalmente sigue siendo el tiempo que implicaría poder calcular cual sería la jugada mas optima, pero aún se pueden hacer algunas cosas para reducir el tiempo de búsqueda. Observemos la figura 2. El valor para el nodo A es 3, y el primer valor encontrado para el subárbol que empieza en el nodo B es 2. Así que como el nodo B está en un nivel MIN, sabemos que el valor seleccionado para el Nodo B debe ser menor o igual a 2. Pero también sabemos que el nodo A tiene el valor 3 y los nodos A y B comparten el mismo padre en un nivel MAX. Esto significa que la ruta del juego que comienza en el nodo B no se seleccionaría porque 3 es mejor que 2 para el nodo MAX.

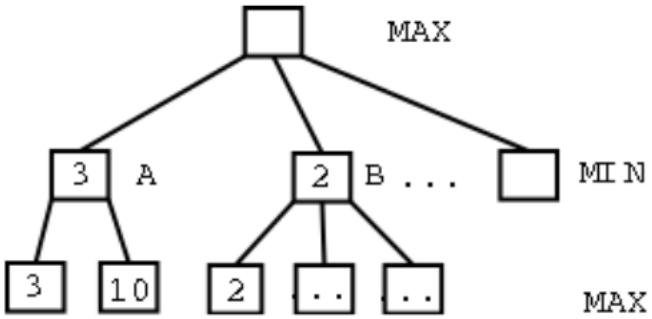


Figure 2: Algoritmo alpha-beta

Esto significa que a veces la búsqueda puede ser abortada porque descubrimos que el subárbol de búsqueda no nos llevará a ninguna respuesta viable.

Esta optimización se conoce como alpha-beta minimax y el algoritmo es el siguiente:

- Tenga dos valores pasados alrededor de los nodos de árbol:
  - el valor alfa que contiene el mejor valor MAX encontrado;
  - el valor beta que contiene el mejor valor MIN encontrado.
- En el nivel MAX, antes de evaluar cada ruta secundaria, compare el valor devuelto con el de la ruta anterior con el valor beta. Si el valor es mayor entonces abortamos la búsqueda del nodo actual;
- En el nivel MIN, antes de evaluar cada ruta secundaria, compare el valor devuelto con el de la ruta anterior con el valor alfa. Si el valor es menor entonces abortamos la búsqueda del nodo actual.

# 4 Explicación de Minimax básico y optimizado por corte

## 4.1 Explicación General

1. Generación del árbol de juego. Se generarán todos los nodos hasta llegar a un estado terminal o determinando una profundidad concreta.
   - Vamos aplicando el algoritmo por un número fijo de iteraciones hasta alcanzar una determinada profundidad. En estas aplicaciones la profundidad suele ser el número de movimientos o incluso el resultado de aplicar diversos pasos de planificación en un juego de estrategia.

2. Cálculo de los valores de la función de utilidad para cada nodo terminal.
    - Para cada resultado final, cuan beneficioso me resulta si estamos en MAX o cuanto me perjudicará si estamos en MIN.
3. Calcular el valor de los nodos superiores a partir del valor de los inferiores. Alternativamente se elegirán los valores mínimos y máximos representando los movimientos del jugador y del oponente, de ahí el nombre de Minimax. 4 . Elegir la jugada valorando los valores que han llegado al nivel superior.

## 4.2 Minimax Básico:

Se tiene el siguiente grafo:

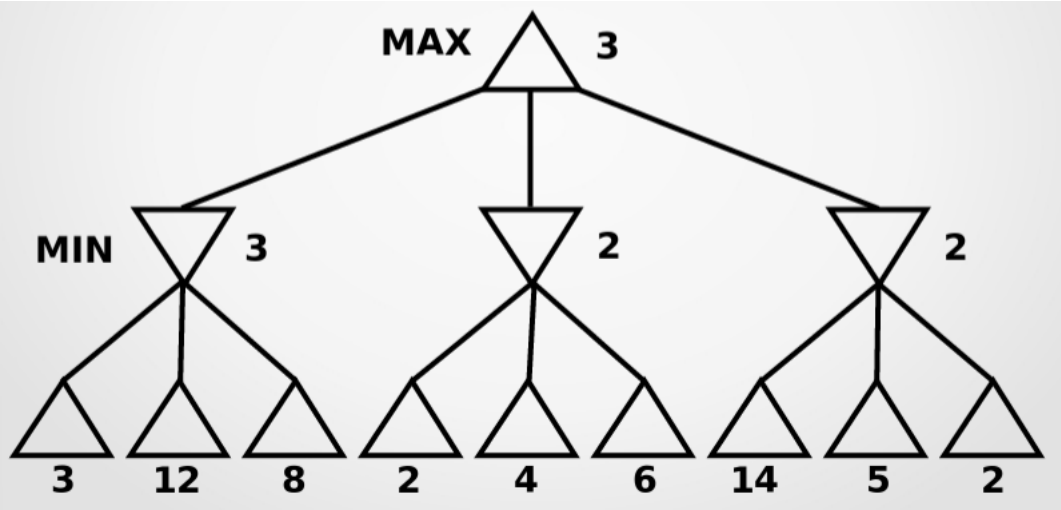


Figure 3: Minimax Básico

- Podemos notar que cada unos de los score se encuentran en los nodos hoja y que el juego posee ramificación 3.
- El segundo nivel es MIN entonces los valores asociados a estos nodos será el minimo valor de sus hijos. Quedando 3, 2 y 2 respectivamente en cada uno de esos nodos.
- Ahora subiendo un nivel y encontrándose ya en nodo raiz vemos que es MAX, entonces su valor asociada el maximo valor de sus nodos hijos, quedando como valor optimo 3.

## 4.3 Minimax AlphaBeta

Tomando el mismo arbol de busquedad explicaremos espicificamente la optimación aphabeta mediante cortes.

- Recordemos que recorremos el arbol por profundidad desde la izquierda hacia la derecha.
- En la primera rama del nodo raiz tenemos un nodo que tiene hojas: 3,12,8. Siendo MIN se elegirá primero el 3 estableciendo el beta del nodo en 3, luego 12 y 8 pero como estos valores son mayores no realiza ningún cambio en el nodo padre, por ultimo como ya se evaluó todo las hojas del nodo el unico valor minimo llega a ser 3, entonces ese valor se pasa al valor alpha de su nodo padre(en este caso el nodo raiz), esto quiere decir que como minimo puede ser 3 y solo espera valores que sean mayores a 3 para tomarlos en cuenta. Y se verifica al siguiente nodo hijo de la raiz.
- Seguimos en un nodo MIN y vemos que su primera hoja es 2, entonces se establece el beta en 2, esperando valores que sean menores a 2, es aqui donde ocurre la poda pues, al ser minimo y esperar valores menores a 2 ya no es necesario recorrer mas hojas pues si son menos solo se alejaran de la solucion del nodo raiz que espera valores mayores a 3, y sabemos que siendo 2 y el nodo MIN, no habría ningun cambio si los otros nodos son mayores a 2 pues solo espera valores menores, entonces ya no se evalua las siguiente ramas.
- Entonces se pasa a la ultima rama del nodo raiz para ser evaluada y asi sucesivamente.

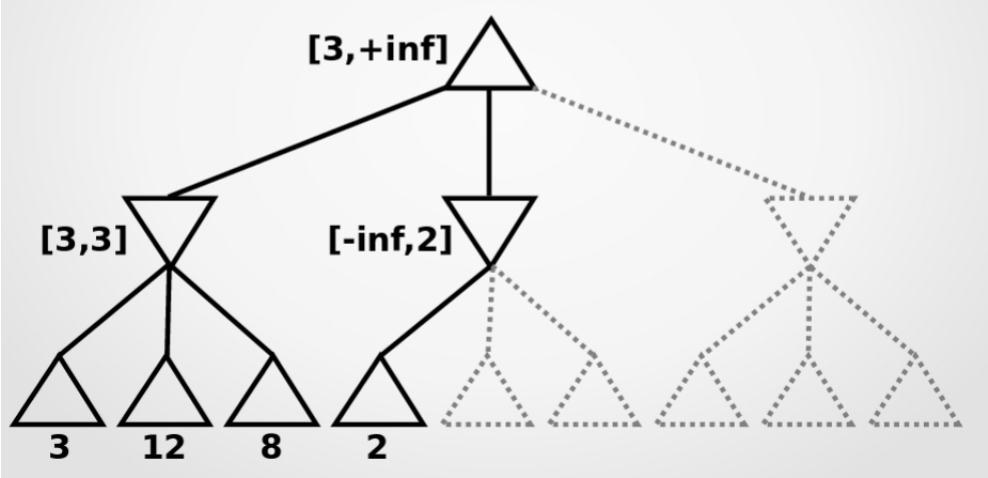


Figure 4: Minimax AlphaBeta

# 5 Implementación de la función en C++

El algoritmo MinMax no es una gran implementación. De hecho, debo mencionar que lo mejor de todo es que funciona. Sin embargo, creo que presenta una manera que el algoritmo podría ser implementado y como un ejemplo es lo suficientemente bueno.

## 5.1 Minimax Básico

```
int Minimax(int depth, int nodeIndex, bool isMax, vector<int> scores){
    // Condición de terminación. Es decir, el nodo hoja es alcanzado
    if (depth == h)
        return scores[nodeIndex];

    // Si el desplazamiento actual es maximizar, se busca el valor máximo alcanzable
    if (isMax)
       return max(Minimax(depth+1, nodeIndex*2, false, scores),
            Minimax(depth+1, nodeIndex*2 + 1, false, scores));

    // Si el movimiento actual es Minimizar, se busca el valor mínimo alcanzable
    else
        return min(Minimax(depth+1, nodeIndex*2, true, scores),
            Minimax(depth+1, nodeIndex*2 + 1, true, scores));
}
```

## 5.2 Minimax AlphaBeta

```
int AplphaBetaPruning(int depth, int nodeIndex, bool maximizingPlayer,
vector<int> values, int alpha, int beta){
    // Condición de terminación. Es decir, el nodo hoja es alcanzado
    if (depth == h)
        return values[nodeIndex];

    if (maximizingPlayer){
        int best = MIN;
        // Recorre los nodos hijos de izquierda a derecha
        for (int i=0; i<2; i++){
            int val = AplphaBetaPruning(depth+1, nodeIndex*2+i,false, values, alpha, beta);
            best = max(best, val);
            alpha = max(alpha, best);
```

```
            // Alpha Beta Pruning
            if (beta <= alpha)
                break;
        }
        return best;
    }else{
        int best = MAX;
        // Recorre los nodos hijos de izquierda a derecha
        for (int i=0; i<2; i++){
            int val = AplphaBetaPruning(depth+1, nodeIndex*2+i,true, values, alpha, beta);
            best = min(best, val);
            beta = min(beta, best);
            // Alpha Beta Pruning
            if (beta <= alpha)
                break;
        }
        return best;
    }
}
```

# 6 Tic Tac Toe

A continuación se tiene la aplicación de Minimax en el juego tictactoe, podrás visualizar el algoritmo mediante el juego de la computadora vs computadora, o uno en el cual tu oponente sea el algoritmo Minimax(computadora).

## 6.1 Decisión en cada turno

En el algoritmo, la ejecución por parte de la computadora será la siguiente:

- Se evaluará cada unos de las posibles posiciones disponibles y jugará en el espacio donde la ganancia se optima.

```
void computerMoveO(int board[9]) {
    int move = -1;
    int score = -2;
    int i;
    for(i = 0; i < 9; ++i) {
        if(board[i] == 0) { // si esta vacio
            board[i] = 1;
            int tempScore = -minimax(board, -1);
            board[i] = 0;
            if(tempScore > score) {
                score = tempScore;
                move = i;
            }
        }
    }
    board[move] = 1;
}
```

```
void computerMoveX(int board[9]) {
    int move = -1;
    int score = -2;
    int i;
    for(i = 0; i < 9; ++i) {
        if(board[i] == 0) { // si esta vacio
            board[i] = -1;
            int tempScore = -minimax(board, 1);
            board[i] = 0;
            if(tempScore > score) {
                score = tempScore;
                move = i;
            }
        }
    }
    board[move] = -1;
}
```

## 6.2 Método Minimax

La función evaluará cada unas de los movimientos posibles tanto del propio jugador como su oponente que se ejecutaran desde el estado actual hasta el ultimo movimiento.

```
int minimax(int board[9], int player) {
    int winner = win(board);
    if(winner != 0) return winner*player;

    int move = -1;
    int score = -2;
    int i;
    for(i = 0; i < 9; ++i) {
        if(board[i] == 0) {
            board[i] = player;
            int thisScore = -minimax(board, player*-1);
            if(thisScore > score) {
                score = thisScore;
                move = i;
            }
            board[i] = 0;
        }
    }
    if(move == -1) return 0;
    return score;
}
```

## 6.3 Ejecución