

Práctica de PROP



Joan Manuel Ramos Refusta

Àlex Aguilera Martínez

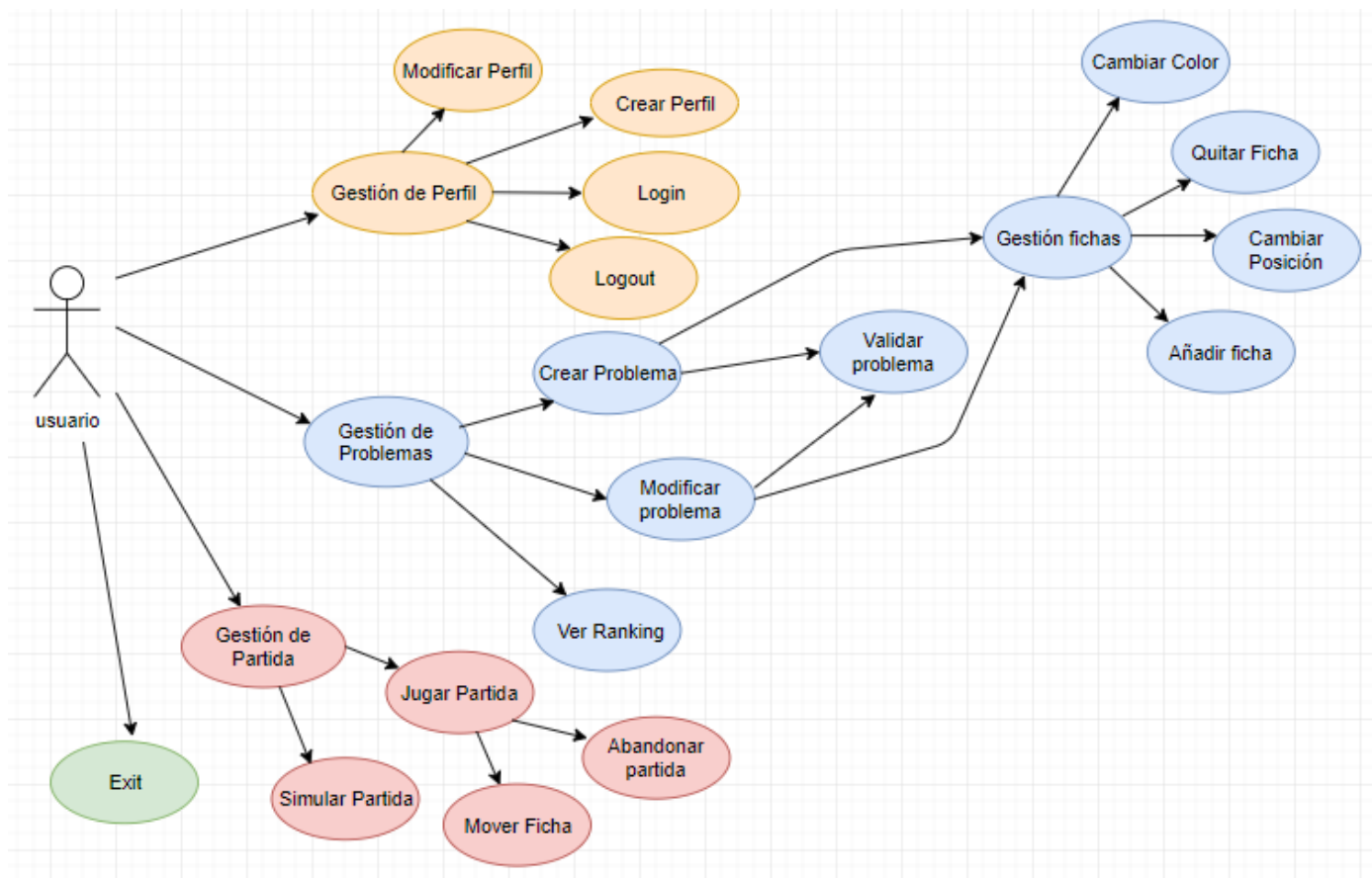
Jordi Santacreu Carranco

Versión 2.0

Índice

1-Diagrama casos de uso.....	3
2-Diagrama estático.....	7
3-Relación clase con miembros del grupo	9
4-Breve descripción de los algoritmos y estructuras.....	10
4.1-Arraylist	10
4.2-List.....	10
4.3-Map.....	10
4.4-MiniMax normal	11
4.5- MiniMax Puro	12
4.6-AlphaBetaMiniMax.....	13

1-Diagrama casos de uso



Gestión perfil

El usuario puede gestionar su perfil, creando un nuevo perfil, iniciar sesión en una cuenta existente, modificando el perfil, eliminándolo o cerrar la sesión iniciada.

Modificar perfil

El usuario, una vez haya iniciado su sesión, puede modificar la contraseña, siempre que la contraseña introducida no sea igual a la que tenía previamente. Si no se produce ningún error durante proceso, la aplicación nos dirá que el cambio se ha producido correctamente.

Crear perfil

El usuario crea un nuevo perfil válido, se introduce un nombre y una contraseña válidos, siempre que el nombre de usuario no haya sido creado previamente.

Login

El usuario introduce el nombre y contraseña de su perfil creado previamente y si los datos son correctos iniciará sesión, en caso contrario, la aplicación nos informará de que alguno de los datos es erróneo y que debe registrarse previamente para loguearse.

Logout

El usuario desconectará su sesión y accederá de nuevo al menú de inicio de sesión.

Eliminar perfil

El usuario, una vez haya iniciado su sesión, puede eliminar su perfil, destruyendo así todos los datos registrados en la aplicación (ranking y problemas creados incluidos). Y al acabar se le informa de ello.

Gestión de problemas

El usuario puede gestionar los problemas, creando uno nuevo, modificándolo siendo el creador, puede ver el ranking de cada uno de los problemas, validar un problema cuando lo haya creado o modificado.

Crear problema

El usuario crea un nuevo problema desde cero, escogiendo el número de movimientos necesarios para ganar el problema y la posición y color de las fichas en el tablero con ayuda de la interfaz. El problema nuevo no debe de existir previamente y se guardarán los datos del creador en dicho problema después de validar que tiene solución en los movimientos que ha introducido anteriormente.

Gestión fichas

El usuario puede gestionar las fichas en el momento de crear el problema o de modificarlo. Podrá cambiar el color, eliminarla, cambiar su posición y añadir una nueva para ir formando su problema correctamente.

Cambiar color

El usuario puede cambiar el color de una ficha que tiene en una posición determinada del problema que está creando o modificando.

Quitar ficha

El usuario puede eliminar una ficha que tiene en una posición determinada del tablero del problema que está creando o modificando.

Cambiar posición

El usuario puede cambiar la posición una ficha que tiene en una posición determinada del tablero del problema que está creando o modificando a su voluntad.

Añadir ficha

El usuario puede añadir una ficha nueva en una posición determinada del tablero del problema que está creando o modificando a su voluntad.

Modificar problema

El usuario modifica un problema siempre y cuando éste sea su creador y ningún usuario lo haya jugado aún. Cuando el usuario lo haya modificado, si es validado correctamente, se introducirá en la lista de problemas jugables del programa. En caso contrario, se le informará al usuario de que su problema no es jugable y por tanto no será introducido en la lista de problemas.

Validar problema

El usuario, una vez haya creado o modificado un problema, es validado para ver si es un problema jugable en el número de movimientos que ha introducido el usuario.

Ver ranking

El usuario puede acceder a los rankings de puntuación de cada problema existente. En ella verá todos los usuarios que han superado el problema ordenados de forma descendente según el tiempo obtenido al jugar.

Gestión partida

El usuario puede elegir entre simular una partida (entre máquinas) o jugar partido entre dos jugadores o contra una máquina.

Jugar partida

El usuario empieza una partida, seleccionando un problema a resolver, en caso de querer jugar contra otro jugador, el segundo jugador deberá de introducir sus datos. Una vez se haya comprobado que los datos sean correctos, la aplicación preguntará el problema a jugar y empezará la partida. Una vez empezada, los usuarios pueden decidir quien defiende o ataca.

En caso de que queramos jugar contra la máquina, el usuario

escogerá el problema a jugar y la dificultad de la máquina con la que jugará.

Una vez acabada la partida, si el jugador ha superado el problema, el jugador será introducido o actualizado, si hace un tiempo mejor, en el ranking de ese problema.

Mover Ficha

El usuario selecciona una ficha para moverla en una casilla del tablero. Esa casilla debe estar dentro de los posibles movimientos según el tipo de ficha que sea. En caso de no estarlo, se le informará al usuario de que ese movimiento no es posible.

Simular partida

El usuario selecciona dos máquinas distintas o iguales y K problemas a superar por estas máquinas. La simulación hará que estas dos máquinas ataquen y defiendan los K problemas, de los problemas que hayan superado, se hará una evaluación y saldrá la máquina ganadora que haya superado más problemas. En caso de empate, se le informará al usuario.

Abandonar partida

El usuario, dentro de una partida, podrá abandonarla, volviendo al menú para escoger otro problema a jugar.

Exit

Se sale de la aplicación.

2-Diagrama estático

- **Jugador:** Es la clase implementada para jugar una partida, es una clase abstracta que tiene como subclases usuario y máquina necesarios para diferenciar si juega uno u otro.

Métodos importantes

- **Pair Getnextmove(Problema p):** Con esta función obtenemos donde se va a mover nuestro jugador. Dependiendo de si es Usuario o Máquina, el procedimiento es distinto. Es una operación abstracta.
- **Usuario:** Subclase de la clase jugador implementada para jugar por personas en nuestro juego de ajedrez y para gestionar los rankings de los problemas. Tiene como atributos nombre (heredada de jugador) y una contraseña para poder iniciar sesión.

Métodos importantes

- **Pair Getnextmove(Problema p) :** Con esta función obtenemos donde se va a mover nuestro usuario, pidiéndole las coordenadas por terminal del movimiento que desea hacer.
- **Máquina:** Subclase de la clase jugador implementada para la IA. Usada para la evaluación de K problemas y para que el usuario intente vencerla. Tiene como atributos nombre (heredada de jugador), dificultad (fácil o difícil) y una profundidad de cara al minimax. (algoritmo usado para la IA).

Métodos importantes

- **Get dificultad:** Devuelve la dificultad de la máquina. Puede ser 1 (fácil) o 2 (difícil)
- **Pair Getnextmove(Problema p):** Con esta función obtenemos donde se va a mover nuestra máquina, en este caso, se tiene en cuenta el mejor movimiento que devuelve nuestro algoritmo Minimax. .
- **Ranking:** Es la clase implementada para guardar todos aquellos usuarios que superen un problema. Hay un ranking por problema. Este ranking ordena según el tiempo logrado por el usuario de manera que se tendrá siempre ordenado del menor al máximo tiempo.

Métodos importantes

- **Ordenar():** Con esta función ordenamos nuestro hash map , esta ordenación se hace por el valor del map i de manera ascendente , es decir , del más pequeño al más grande.
- **EliminarUsuario(string nom) :** Con esta función eliminamos del map, el nodo el cual tiene como key la variable nom. Necesaria para poder actualizar nuestro ranking.
- **Partida:** Es la clase implementada para poder jugar a un problema el cual se carga cuando iniciamos la partida. Esta cuenta con dos jugadores que pueden ser dos máquinas, dos jugadores o jugador contra máquina. El objetivo es lograr hacer mate en un problema en el número de movimientos que establece el mismo.

Métodos importantes

- **mover(boolean color,String cord1,String cord2):** Con esta función hacemos un

movimiento, dado dos coordenadas. Nos dice si el movimiento realizado es correcto o si la coordenada introducida es incorrecta con un booleano.

Problema: Es la clase implementada para cargar nuestro tablero mediante el fen que proporcione el usuario. Además, tiene otros atributos como son el número de movimientos para hacer mate, el turno inicial (obtenido del fen) y el ranking de los usuarios que han intentado el problema. Tiene también, métodos necesarios para el momento de jugar como son el mate/checkmate y el mover ficha.

Métodos importantes

- actualizarRanking(String nombre , double tiempo): Actualiza el tiempo del usuario cuando haya pasado el problema .
- introducirElemento(String nombre,double tiempo) : Introduce una key y un valor al ranking.
- printTablero(): Con esta función imprimos la matriz generada por el fen introducido anteriormente.
- matrixToFen(): Con esta función convertimos la matriz de fichas a un string fen.
- FenToMatrix(string fen): Con esta función convertimos el fen que le pasamos como parámetro a la matriz de fichas de nuestra clase.
- moveFicha(String s1, String s2): Dadas dos posiciones , mueve la ficha de coord. C1 a C2 siempre y cuando c2 se pueda acceder y no haya una ficha igual al color al que movemos y este dentro del tablero . Si hay una ficha rival en C2, nos la comemos.

- **Ficha:** Es la clase implementada para hacer de superclase de todos los tipos de fichas del ajedrez y para fijar los posibles movimientos de las mismas. Es una clase abstracta ya que los diferentes tipos tienen posibles movimientos diferentes.

Métodos importantes

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c):Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos que puede tener una ficha según el tipo que sea .

- **Peón:** Subclase de la clase ficha que implementa los posibles movimientos de una ficha peón del ajedrez: moverse dos en caso de estar en la segunda/penúltima fila, avanzar uno o comer otra ficha en diagonal.

Métodos importantes

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c): Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos
- **Reina:** Subclase de la clase ficha que implementa los posibles movimientos de una ficha reina del ajedrez: básicamente puede hacer la suma de los movimientos de un alfil y de una torre, es decir, se puede mover tanto de manera horizontal como vertical o diagonal. Es la ficha más poderosa.

Métodos importantes

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c): Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos que puede tener una reina.

- **Caballo**: Subclase de la clase ficha que implementa los posibles movimientos de una ficha caballo del ajedrez: se puede mover dos casillas horizontalmente y una casilla verticalmente o dos casillas en posición vertical y una horizontal cuadrada. Es decir, haciendo Ls por el tablero.

Métodos importantes

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c): Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos que puede tener un caballo.
- **Alfil**: Subclase de la clase ficha que implementa los posibles movimientos de una ficha alfil del ajedrez: se puede mover sobre el tablero en una línea recta diagonal.

Métodos importantes

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c): Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos que puede tener un alfil.
- **Torre**: Subclase de la clase ficha que implementa los posibles movimientos de una ficha alfil del ajedrez: se puede mover en línea recta con movimientos horizontales y verticales sobre el tablero.

Métodos importantes

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c): Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos que puede tener una torre.
- **Rey**: Subclase de la clase ficha que implementa los posibles movimientos de una ficha rey del ajedrez. Es la pieza fundamental del tablero solo se puede desplazar en una casilla tanto en vertical, horizontal o en diagonal. Si el rey puede ser comido por otra pieza es jaque y si no puede hacer movimiento para evitarlo es jaque mate.

Métodos importantes

- ArrayList<Coordenada> posiblesMovimientos(Problema p, Coordenada c): Dado un problema y una coordenada cualquiera dentro del tablero del problema, devuelve todos los posibles movimientos que puede tener un rey.

Hemos implementado además algunas clases extra como son **Minimax**, **Coordenada** y **Pair** por necesidad y para el correcto funcionamiento de nuestro programa. Aunque Coordenada y Pair sean clases muy similares, hemos optado en implementar las dos, debido a que Coordenada necesita dos métodos que son StringToCoordenada y CoordenadaToString para representar las coordenadas más fácilmente.

3-Relación clase con miembros del grupo

La relación entre las clases con los miembros del grupo ha sido la siguiente:

- Àlex: **Primera entrega**: Ha hecho todas las clases relacionadas con las fichas del tablero y la clase MiniMax, la cual encontramos el algoritmo que usamos para jugar contra otro usuario (IA), la clase Máquina y sus respectivos drivers.

Segunda entrega: Ha hecho los dos controladores de Persistencia, VistaPartida, CtrlPresentacionJugar, MinimaxAlphaBeta, CtrlPresentacionJugarMaq y CtrlPartida.

- Joan: **Primera entrega:** Ha hecho la clase Problema, la clase Jugador, los controladores de problemas y usuarios y también sus respectivos drivers. **Segunda entrega:** Ha hecho CtrlPresentacionCtrlProblemas, CtrlPresentacionProblema, VistaCrearModificarProblema, VistaMenu, VistaPreview, VistaProblemaRanking, VistaProblemasJug, VistaProblemasMaq, VistaProblemasUsuarios, VistaProblemasVS y VistaProfile.

- Jordi: **Primera entrega:** Ha hecho la clase Ranking, Partida, Jugador, el JUnit y sus respectivos drivers. Al igual que las clases extra implementadas Coordinada y Pair con sus drivers. **Segunda entrega:** VistaRegistro, VistaRanking, VistaInicio, VistaGuestLog, MinimaxPuro, CtrlPresentacionUsuarios y CtrlPresentacionRanking.

4-Breve descripción de los algoritmos y estructuras

4.1-Arraylist

Es una clase que permite almacenar datos en memoria de manera similar a la de un array con la ventaja de que es una estructura flexible que permite añadir elementos sin tener en cuenta su tamaño. Permite añadir y eliminar atributos del ArrayList siempre que el usuario quiera. Algunas funcionalidades; para añadir un nuevo elemento se hace con array.add(variable) y si queremos eliminar un elemento lo haremos mediante array.remove(atributo), para saber el tamaño del ArrayList en un momento dado se usa array.size()...

4.2-List

Son un tipo de estructura que permite almacenar grandes cantidades de datos. Proporciona control sobre la posición en la que puede insertar un elemento y permite acceder a los elementos por su índice y también buscar elementos. Algunas funcionalidades; para añadir elementos haremos listnom.add(elemento) y para eliminar listnom.delete(elemento). Es, además, una colección ordenada motivo por el cual ha sido usada en el ranking para ordenar.

4.3-Map













El map es una estructura ordenada que nos permite almacenar pares "clave/valor"; de tal manera que para una clave solamente tenemos un valor. La hemos utilizado tanto para hacer el ranking en nuestro programa como los CtrlProblemas y CtrlUsuarios. Es una estructura que permite ordenar sus claves de forma automática si lo deseamos, pero en este caso tiene la peculiaridad que también permite ordenar por valor mediante un comparador, esta característica nos ha sido bastante útil en el momento de implementar nuestro ranking.

4.4-MiniMax normal

Es el algoritmo utilizado en esta práctica para poder implementar nuestra máquina (IA). El algoritmo MiniMax es el algoritmo más utilizado para problemas con exactamente 2 adversarios y movimientos alternos. Básicamente, consiste en identificar a cada jugador como MIN y como MAX, este último es el que siempre inicia el juego, y marcar como objetivo encontrar el mejor movimiento que puede hacer MAX (máquina) dado un estado particular en el juego, suponiendo que el contrincante (MIN) escogerá siempre el peor movimiento para que MAX no le gane.

Es necesario una función de evaluación heurística que devuelva valores elevados para las buenas situaciones y valores bastante peores para las situaciones favorables para el oponente, de esta manera cada movimiento tendrá un valor numérico y podremos escoger el mejor de todos los movimientos.

Nosotros, hemos escogido los siguientes valores para cada una de las fichas de nuestro tablero:

	10		-10		50		-50
	30		-30		150		-150
	30		-30		900		-900

La función heurística es la siguiente: (aplicada en el caso base, cuando la profundidad = 0)

```
int evaluationBoard(Problema p) {  
    int totalEvaluation = 0;  
    for (int i = 0; i < 8; i++) {  
        for (int j = 0; j < 8; j++) {  
            totalEvaluation = totalEvaluation + getPieceValue(p.getFicha(i,j));  
        }  
    }  
    return totalEvaluation;  
}
```

Como se puede comprobar, simplemente es coger los valores de las piezas en un estado de la partida determinado y sumarlos. De tal manera que, con los valores negativos y positivos, tendremos un valor que determinará un buen movimiento o no para las máquinas.

Además de utilizar esta función heurística, se usa una estrategia DFS de profundidad limitada (ya que partir de 4, se empieza a notar el coste exponencial del algoritmo) para explorar las jugadas posibles. Lo que quiere decir, que, al no poder estudiar todas las posibles situaciones hasta el fin de partida, puede que haga alguna jugada que no sea lo suficientemente buena en problemas de dificultad superior.

El pseudocódigo es el siguiente:

```
Función valorMin(estado) retorna entero
  valor_min:entero;
  si estado_terminal(estado) entonces
    retorna evaluación(estado);
  sinó
    valor_min := +infinito;
    para cada mov en movimientos_posibles(estado) hacer
      valor_min := min(valor_min, valorMax(aplicar(mov, estado)));
    fpara
  retorna valor_min;
fsi
fFunción

Función valorMax(estado) retorna entero
  valor_max:entero;
  si estado_terminal(estado) entonces
    retorna evaluación(estado);
  sinó
    valor_max := -infinito;
    para cada mov en movimientos_posibles(estado) hacer
      valor_max := max(valor_max, valorMin(aplicar(mov, estado)));
    fpara
  retorna valor_max;
fsi
fFunción

Función MiniMax(estado) retorna movimiento
  mejor_mov: movimiento;
  max, max_actual: entero;
  max := -infinito;
  para cada mov en movimientos_posibles(estado) hacer
    max_actual = valorMin(aplicar(mov, estado));
    si max_actual > max entonces
      max = max_actual;
      mejor_mov = mov;
  fsi
  fpara
  retorna mejor_mov;
fFunción
```

+ Estado se entiende como un momento concreto en la partida (posición de las fichas de una manera concreta).

Explicación básica del pseudocódigo:

- Se inicia un recorrido en profundidad del árbol del juego hasta una profundidad limitada
- Dependiendo del nivel donde se encuentre se llama a una función que obtiene el valor máximo o mínimo de la evaluación de los descendientes
- El recorrido se inicia con la jugada del jugador MAX (en nuestro caso, los dos pueden iniciar)
- La función evaluacion(estado) (la función heurística) es la misma para los dos jugadores y retorna el valor de la función de evaluación para el estado actual en el que se encuentra
- Estado_terminal() determina si ha llegado al caso terminal, es decir al checkmate o a la profundidad máxima que puede llegar. (en nuestro caso, solo evaluamos este último)
- Para terminar, el algoritmo retorna el mejor movimiento, teniendo en cuenta los valores recogidos en la función Minimax() que los obtiene recursivamente.

Esta ha sido la opción escogida de cara a la máquina de nivel fácil. No hemos incluido nada más para mejorar la heurística, a diferencia con el AlphaBeta. Anteriormente, el minimax estaba implementado con una sola función para MIN y MAX, como mejora, ya que no era muy eficiente, las hemos incluido y la evaluación del mejor movimiento ya no debe recorrer un vector, que no era necesario y poco eficiente.

Hemos implementado para la segunda entrega además del AlphaBeta, un Minimax Puro para la validación del problema, cuya implementación no tiene heurística.

4.5 –MiniMax Puro

La idea de este minimax es completamente distinta. Lo que buscamos en este caso es poder validar que en un problema cualquiera se llegue a hacer checkmate(), es decir ganar al contrincante, en N movimientos determinados por el usuario al momento de crear un problema.

Para ello, no es necesario una heurística como en el caso anterior, ya que lo único que

buscamos es la posibilidad de hacer el checkmate en esos movimientos y si es posible guardar el problema como correcto. Simplemente, al llegar profundidad $2*N-1$, retornar +1 en caso de checkmate o -1 en caso contrario. Una idea similar a la que se usa en un minimax para el tres en raya.

4.6-AlphaBetaMiniMax

La poda alfa-beta es un método de optimización del algoritmo minimax que nos permite ignorar algunas ramas en el árbol de búsqueda de tal manera que mejoramos el rendimiento de nuestro algoritmo. Esto es debido a que dejamos de mirar posibles posiciones de las piezas ya que tenemos el mejor movimiento y no hace falta.

Este consiste en una técnica llamada ramificación y poda con la cual abandona una solución parcial en el momento que se sepa que hay otra solución mejor.

Concretamente, se guardan dos valores denominados alfa y beta, de ahí el nombre.

Alfa representa la cota inferior del valor que puede asignarse en último término a un nodo de MAX, y mientras que beta representa la cota superior del valor que puede asignarse como última opción en un nodo de MIN.

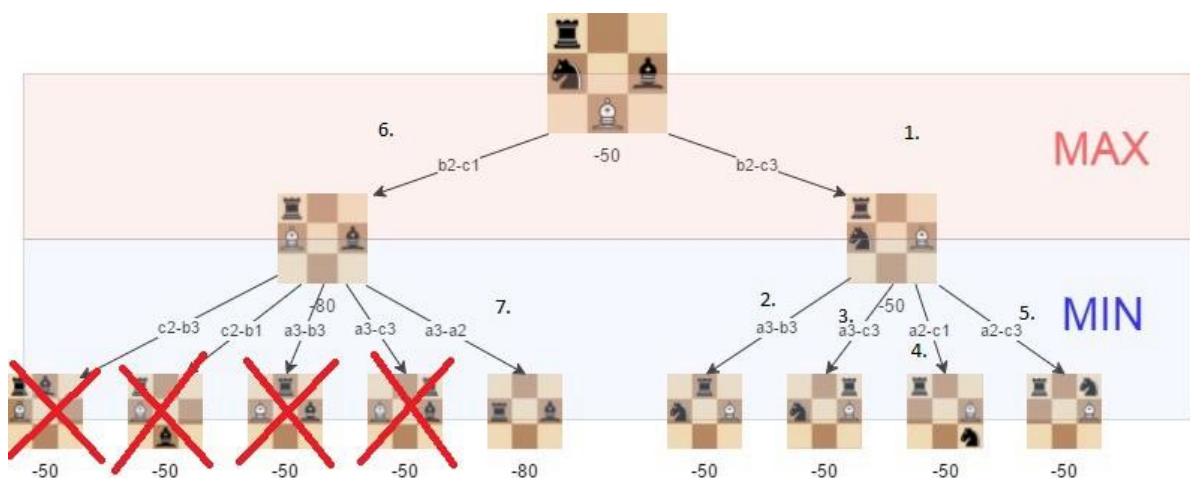
La función minimax (minimaxDecision, en nuestro código) se mantiene igual y cambiaría las funciones max y min, ya que debe evaluar alfa y beta de la siguiente manera:

```
si  $\beta \leq \alpha$ 
    romper (* poda  $\alpha$  *)
```

```
si  $\beta \leq \alpha$ 
    romper (* poda  $\beta$  *)
```

El primer if debería en la función min mientras que el segundo en la función max. Donde alfa y beta es lo que retorna max() o min() en respectivas funciones.

De esta manera, conseguimos ir podando el espacio de búsqueda de nuestro árbol. (como se puede ver en el siguiente ejemplo)



A parte de esta mejora respecto al minimax normal, hemos introducido una mejora en la función que evalúa el tablero (los valores de la heurística son los mismos) usando las siguientes tablas:



[-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[-3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[-2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
[-1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
[2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 2.0, 2.0],
[2.0, 3.0, 1.0, 0.0, 0.0, 1.0, 3.0, 2.0]



[-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
[-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0],
[-1.0, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -1.0],
[-0.5, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -0.5],
[0.0, 0.0, 0.5, 0.5, 0.5, 0.5, 0.0, -0.5],
[-1.0, 0.5, 0.5, 0.5, 0.5, 0.5, 0.0, -1.0],
[-1.0, 0.0, 0.5, 0.0, 0.0, 0.0, 0.0, -1.0],
[-2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]



[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.5],
[-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5],
[-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5],
[-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5],
[-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5],
[-0.5, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.5],
[0.0, 0.0, 0.0, 0.5, 0.5, 0.0, 0.0, 0.0]



[-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
[-1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0],
[-1.0, 0.0, 0.5, 1.0, 1.0, 0.5, 0.0, -1.0],
[-1.0, 0.5, 0.5, 1.0, 1.0, 0.5, 0.5, -1.0],
[-1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, -1.0],
[-1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0],
[-1.0, 0.5, 0.0, 0.0, 0.0, 0.0, 0.5, -1.0],
[-2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]



[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
[-4.0, -2.0, 0.0, 0.0, 0.0, 0.0, -2.0, -4.0],
[-3.0, 0.0, 1.0, 1.5, 1.5, 1.0, 0.0, -3.0],
[-3.0, 0.5, 1.5, 2.0, 2.0, 1.5, 0.5, -3.0],
[-3.0, 0.0, 1.5, 2.0, 2.0, 1.5, 0.0, -3.0],
[-3.0, 0.5, 1.0, 1.5, 1.5, 1.0, 0.5, -3.0],
[-4.0, -2.0, 0.0, 0.5, 0.5, 0.0, -2.0, -4.0],
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]



[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0],
[1.0, 1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0],
[0.5, 0.5, 1.0, 2.5, 2.5, 1.0, 0.5, 0.5],
[0.0, 0.0, 0.0, 2.0, 2.0, 0.0, 0.0, 0.0],
[0.5, -0.5, -1.0, 0.0, 0.0, -1.0, -0.5, 0.5],
[0.5, 1.0, 1.0, -2.0, -2.0, 1.0, 1.0, 0.5],
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Las tablas de las fichas negras, son justamente las mismas inversas. Con estas tablas lo que conseguimos es que la función evaluación tenga en cuenta la posición de las fichas en el tablero lo que puede mejorar un poco el juego de la máquina. Ya que, por ejemplo, un caballo en el centro del tablero es mejor (porque tiene más opciones y, por lo tanto, es más activo y puede hacer más) que un caballo en el borde del tablero donde tiene menos posibles movimientos.

Esta es la opción escogida de cara a la máquina con dificultad difícil. Si hubiéramos tenido más tiempo, nos hubiera gustado haber mejorado este último algoritmo con otras opciones como una evaluación específica del final del juego, ordenar los movimientos que pueden ser mejores para la máquina o evitar el efecto horizonte.