

# Agente Inteligente para SHÖBU usando estrategia Minimax

Presentado por:

Jhon Edinson Cortes Navarro - 202013032

Michael Palacios Gavria - 201356132

Luis Eduardo Henao Padilla - 201667483

Universidad del valle Sede Tuluá

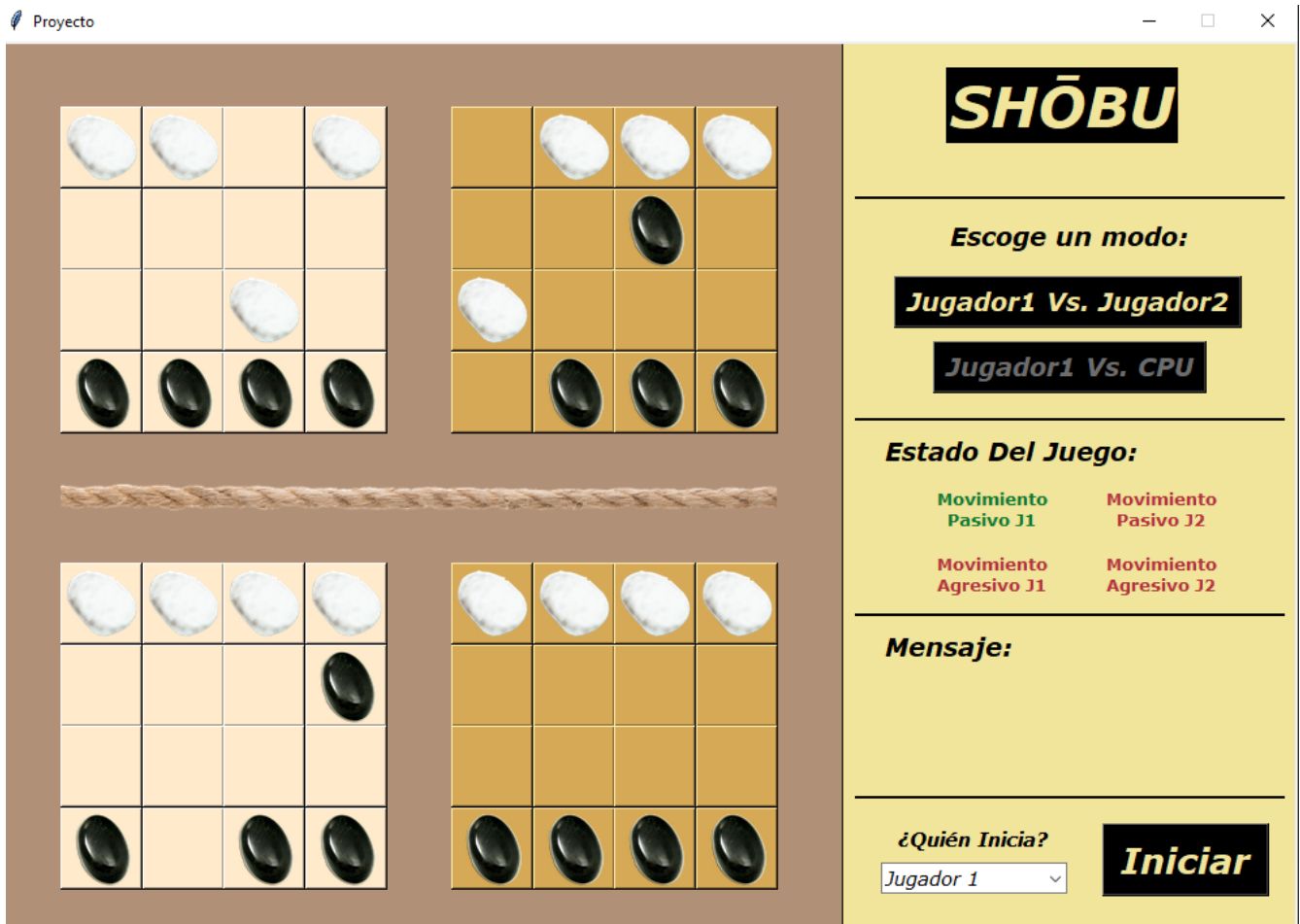
Facultad de Ingeniería

Ingeniería de Sistemas

Introducción a la Inteligencia artificial

13/04/2021

# SHÖBU



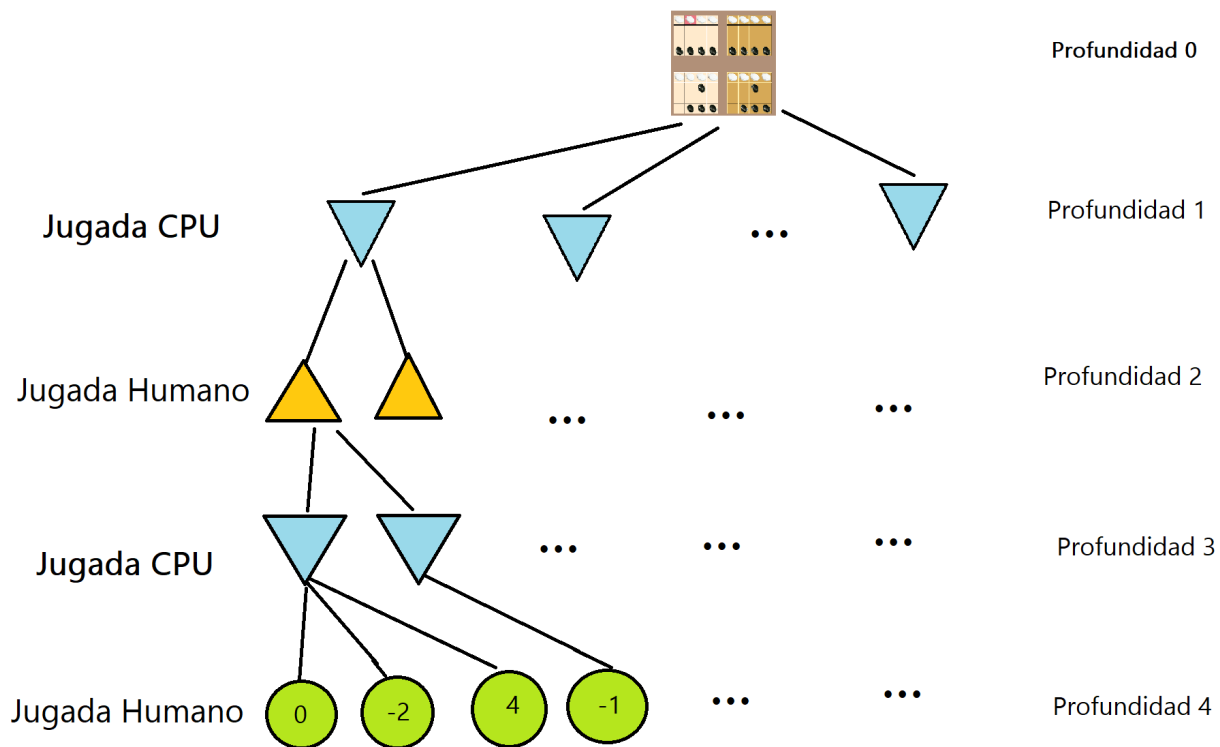
Shobu es un juego de estrategia abstracta elaborado para dos jugadores. El juego cuenta con 4 tablas cuadradas de madera (2 de cada color) y 16 piedras naturales de río para cada jugador, en dos colores, con una cuerda dividiendo el área de juego por la mitad.

## Reglas del juego:

1. Inician las piedras negras dando el primer movimiento al inicio de cada juego, en el lado de la cuerda del jugador que tenga las piedras negras no en el lado contrario de la cuerda.
2. los movimientos se basan en dos movimiento pasivo, con el cual el jugador de turno realiza el movimiento en su lado de la cuerda y el movimiento agresivo lo realiza en lado contrario de la cuerda y tiene que ser el mismo movimiento que se realizó en el movimiento pasivo, mismas casillas recorridas de la piedra y misma dirección.
3. El movimiento pasivo si se realiza en un cuadro claro por lo tanto cuando se realiza el movimiento agresivo tiene que ser en el cuadro oscuro de lado contrario de la cuerda, en el lado del contrincante, lo mismo para el cuadro oscuro si se hizo el movimiento pasivo en un cuadro oscuro por lo tanto el movimiento agresivo se realiza en el cuadro claro en el lado del contrincante.
4. Cuando se realiza el movimiento pasivo y se tiene seleccionada la ficha a mover, solo se puede recorrer uno o dos casillas dentro del cuadro que se escogió ya sea claro u oscuro.
5. los movimientos a realizar son 8 dependiendo del estado de la piedra verificando si se puede hacer el movimiento arriba, derecha, abajo, izquierda, diagonal superior derecha o izquierda, diagonal inferior derecha o izquierda y si la magnitud del movimiento es menor o igual a 2.
6. Estados durante el juego, en esta parte los estados funcionan para cualquiera de los dos jugadores:
  - 6.1. #0: Selección de Piedra negra
  - 6.2. #1: movimiento pasivo jugador con piedras negras
  - 6.3. #2: Selección de Piedra negra
  - 6.4. #3: movimiento agresivo jugador con piedras negras
  - 6.5. #4: Selección de Piedra blanca
  - 6.6. #5: movimiento pasivo jugador con Piedras blanca
  - 6.7. #6: Selección de Piedra blanca
  - 6.8. #7: movimiento agresivo jugador con Piedras blanca
7. Se determina el fin del juego cuando uno de los dos jugadores se queda sin fichas en cualquiera de los cuatro cuadros, solo debe de quedar fichas del mismo color en un cuadro para determinar el fin del juego.

# Construcción del árbol:

El árbol de posibles jugadas de SHÖBU que nuestro equipo de desarrollo implementó en el proyecto está hecho basándonos en los ejemplos vistos durante las clases:



Nuestra estructura de árbol cuenta con un estado inicial (nodo padre), dicho estado puede variar dependiendo de quien inicie la partida (si inicia el jugador humano o la CPU) y obviamente cambiará mediante el transcurso de la partida.

Cada nodo hijo corresponde a una posible jugada de la IA o del jugador humano. Una jugada corresponde a un movimiento pasivo y un movimiento agresivo; cada nodo hijo almacena de la jugada: Los objetos piedra que se están jugando, la posición de la piedra que se jugó la jugada pasiva, el movimiento (magnitud y dirección) de la jugada pasiva, la posición de la piedra que se jugó en la jugada agresivo y los tableros en los que se jugaron las anteriores jugadas; Pero, si un nodo hijo es a su vez un nodo hoja o nodo terminal, estos no almacenan la información anterior, sino que almacenarán principalmente la utilidad de la jugada realizada, esta utilidad resultará útil en el algoritmo minimax a la hora de escoger una jugada.

Nuestro árbol solo llega hasta una profundidad de 4, por lo que es correcto afirmar que todos los nodos hoja que estén en la profundidad 4 almacenarán la utilidad requerida para el algoritmo minimax, pero también cabe recalcar, que para cada nodo diferente al padre se verificará si alguno de los jugadas resulta en la jugada ganadora, ya sea para la IA o para el jugador humano, una vez confirmado el ganador se guarda en la utilidad el valor ( $\infty$ ) o  $-(\infty)$  y no se recorrería más, debido a que es inútil seguir jugando habiendo ya un ganador.


La función recursiva que se encarga de generar esta estructura árbol se llama `jugada()` y se encuentra en el archivo llamado `proyectoFinal()` y más que generar las jugadas, gestiona las posibles jugadas y cómo se almacenan en la estructura de cada nodo.

Durante la implementación del árbol y el algoritmo minimax, observamos un problema y es que el realizar la estructura árbol de una serie de jugadas con la profundidad 4, tardaba demasiado tiempo en realizarse. La solución sencilla sería bajarle la profundidad al árbol, para disminuir los posibles movimientos o tratar de optimizarlo para evitar que ejecute más código del que debería.

### ¿Realmente nuestro árbol recorre todas las posibles futuras jugadas?

Realmente no, al ser demasiadas posibilidades para cada jugada: por ejemplo la probabilidad de mover una ficha es de 576 posibles movimientos válidos en el primer turno es decir 72 posibles movimientos pasivos con sus respectivos movimientos agresivos en las posibles 8 piedras restantes, generando un árbol de crecimiento exponencial.

Para tratar de solucionar este crecimiento acelerado, decidimos que una piedra pasiva solo podría jugar (si la distribución uniforme es la correcta) con la mitad de sus posibles movimientos:

(-2,2)		(0,2)		(2,2)
	(-1,1)	(0,1)	(1,1)	
(-2,0)	(-1,0)		(1,0)	(2,0)
	(-1,-1)	(0,-1)	(1,-1)	
(-2,-2)		(0,-2)		(2,-2)

Posibles Movimientos

```
255 | for movimiento in posiblesMovimientos:
256 |     if random.uniform(0.0,1.0)>=0.5:
257 |         movimientosSeleccionados.append(movimiento)
258 |
```

Las siguientes línea de código toman de la lista de posibles movimientos, un movimiento y mediante un número aleatorio de distribución uniforme generado, se evalúa si el valor generado es mayor o igual a 0.5, si lo cumple, se añade el movimiento a la lista de movimientos Seleccionados, que en sí será la lista que se utilizará para los posibles movimientos de las piedras.

Adicionalmente y para hacer más aleatoria la situación, decidimos revolver las listas que contienen las piedras de cada jugador (humano o IA) y la lista de posibles movimiento. Esto se planteó como una solución a la poca significancia que puede llegar a tener la utilidad en el minimax, es decir, *que una jugadaX no se diferencie de una jugadaY en cuanto a su utilidad*, por lo que muchas jugadas resultaban tener la utilidad apropiada y el algoritmo minimax ciegamente escogía la primera de ellas. Durante las primeras pruebas del algoritmo pudimos notar este defecto, ya que la IA siempre escogía las primeras piedras de las listas de piedras, esto debido a que eran las primeras en las listas.

```
random.shuffle(l1)
random.shuffle(l2)
random.shuffle(posiblesMovimientos)
```

Para revolver las listas usamos el método shuffle de la librería random de python, aunque soluciono en parte el problema de la insignificancia de la utilidad que se podía producir en las jugadas, nos hizo recalcar otro problema con el uso de la utilidad en el árbol y es que a la IA le queda muy difícil seguir de una jugada a otra, es decir, hace un movimiento pasivo y uno agresivo, una jugada que probablemente en la siguiente jugada le genere una ventaja frente al jugador, pero al añadir estos componentes aleatorios (aunque sean necesarios) puede llegar a imposibilitar a la IA de hacer mejores estrategias.

## Estrategia de Solución:

Se usó el algoritmo minimax para seleccionar aquella jugada que pueda proporcionar una ventaja a la IA o para evitar que el jugador humano pueda perjudicarla con un movimiento, para ello implementamos el algoritmo minimax:

```
minimax(arbolI, 4, -math.inf, math.inf, 1)
```

el algoritmo minimax es una función recursiva que recibe como tal: el árbol que contiene todos los posibles movimientos o escenarios futuros de las jugadas de la IA contra el Jugador humano, la profundidad hasta la cual el algoritmo recorrerá el árbol, un valor alfa ( $-\infty$ ), un valor beta ( $\infty$ ) y si la IA maximiza o minimiza (en caso de ser 0 la IA maximiza y en caso de ser 1 la IA minimiza); En un comienzo, como nuestra profundidad está predeterminada con un valor de 4, la IA siempre minimizará, ya que los nodos terminales correspondientes serán los de las jugadas del humano.

Una vez iniciado el algoritmo se verifica si la profundidad en la que se encuentra actualmente es 0 o si el nodo que se está consultando no tiene hijos que explorar, esto como condición de parada para la función y retornar la utilidad de los nodos terminales.

después dependiendo si se maximiza o minimiza en la ejecución de la función actual, se asigna una evaluación máxima ó mínima de  $(-\infty)$  y  $(\infty)$  respectivamente, se iteran todos los nodos hijos de un nodo padre (o sub nodo padre) y se hace un llamado recursivo de la función hasta que este se recorra por completo, todos los nodos hijos del árbol, cada terminal del hijo debería retornar su utilidad, en la primer iteración se almacena la evaluación y se compara con la evaluación máxima  $(-\infty)$  ó mínima  $(\infty)$  dependiendo si se maximiza o minimiza obviamente y en caso de que al realizar la comparación sea un max o un min se almacena en el nodo padre (sub nodo padre) el hijo seleccionado, pero de igual manera se sigue iterando entre sus hijos para encontrar uno que sea mayor o menor comparado con el anterior.

Se utilizan las podas alfa y betas para disminuir el número de iteraciones en los nodos terminales o nodos hijos del árbol, si se está minimizando se verifica si se puede hacer una poda beta, y si se está maximizando se verifica si se puede hacer una poda alfa.

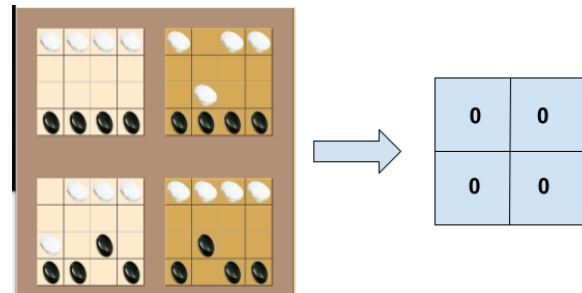
## Función de utilidad

Se definió una función de utilidad que permite capturar el estado de los cuatro tableros en una matriz 2x2 de resultados, de tal manera que las piedras con las que juega la IA son restadas con las del jugador. Esto quiere decir que los tableros con resultados más negativos serán más beneficiosos para la IA.

Sea  $N_{i,j}$ : Número Piedras IA  $i,j$

Sea  $M_{i,j}$ : Número Piedras Jugador  $i,j$

$$\text{Resultados}_{i,j} = -2N_{i,j} + 2M_{i,j}$$



Para hacer más específica la función respecto a las posibles jugadas que podía emprender en cada estado, se agregó un bonus para aquellos estados futuros que produjeran que una ficha del contrincante fuese movida o saliera del tablero.

Sea  $B_{i,j} = 0$

$B_{i,j} = 5$ , si Mueve Ficha Enemiga

$B_{i,j} = 10$ , si Elimina Ficha Enemiga

$$\text{Resultados}_{i,j} = \text{Resultados}_{i,j} - B_{i,j}$$

