Práctica 1
- Inteligencia Artificial -
Minimax

1. Análisis de los resultados

- Tablas de resultados de los algoritmos y sus resultados de velocidad, porcentaje de veces ganadas, profundidad…

2. Problemas encontrados

2.1. Creación del tablero

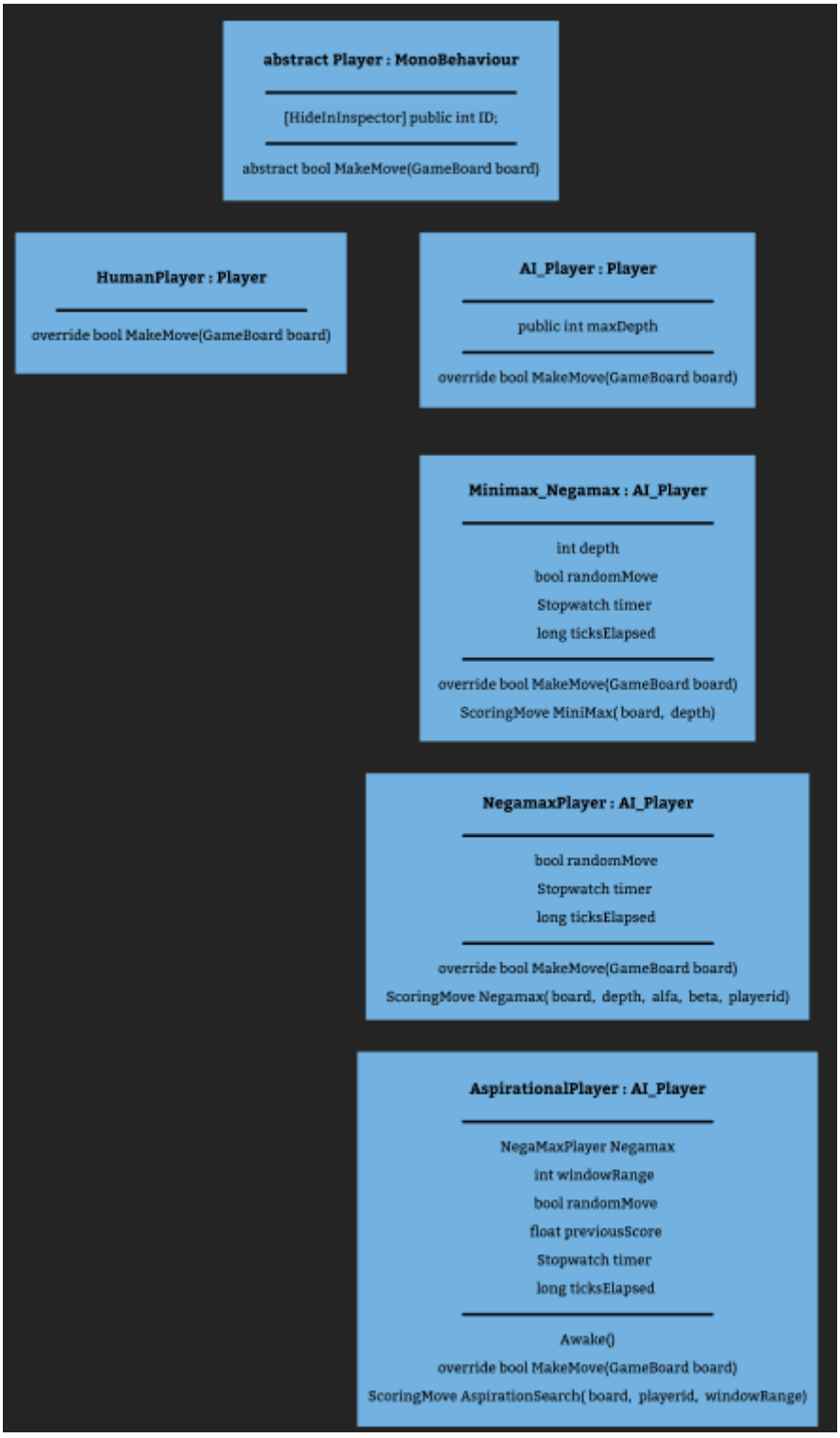
Aunque la creación del tablero pudiera significar un coste de trabajo nulo frente a la nota final no fue del todo sencillo encontrar una forma de ahorrar el mayor tiempo posible mientras se trataba de mantener el conseguir una base sólida sobre la que expandir.

La idea de crear un tablero de cero estuvo presente durante las primeras charlas de como poder llevarlo a cabo pero realmente nos iba a llevar posiblemente demasiado tiempo como para que compensase, por lo que decidimos explorar el marketplace de Unity para ver si alguien podría proporcionarnos la base necesaria para retocar y adueñar.

Encontramos un conecta cuatro con funciones bastante básicas pero útiles, la instanciación de un tablero, comprobaciones de victoria, diferenciación entre las fichas del tablero e incluso una pequeña animación de las piezas cayendo hasta su posición. El código interno no era del todo agradable de mirar, estaba todo comprimido en un solo script y era bastante dependiente unas cosas de otras.

La idea era tratar de encajar esas bases en nuestra idea propia de conecta cuatro y como se debería de distribuir las funcionalidades para poder añadir jugadores automáticos. Nuestra implementación es la siguiente:


Game Manager : MonoBehaviour
GameBoard gameBoard
Player player1
Player player2
GameState gameState
int currentPlayerID
+ StartGame()
IEnumerator GameLoop()
-CheckEnd(playerID, gameState)
-SwitchPlayers()
enum GameState
Playing = 0,
Win,
Draw,
Choosing
Scoring Move
int score;
int column
ScoringMove(int score, int column)
Game Board : MonoBehaviour
int columns, rows
int [,] board
int [,] boardSpaces
int numPiecesToWin
GameObject bluePiece
GameObject redPiece
GameObject fieldPiece
GameManager gameManager
+ GameBoard(Gameboard other)
Awake()
-CreateField()
+List<int> GetPossibleMoves()
+MakeMove(column,playerID)
+ GameBoard CreateTempBoardWithMove(column, playerID)
-MakeTemporalMove(tempBoard, column, playerID)
IEnumerator dropPiece(dropColumn, playerID)
bool IsValidMove(column)
bool FieldContainsEmptyCell()
int GetColumnFromHitPoint(hitPoint)
float GetCellWidth()
int EvaluateBoard(ID)
int CheckForStreak([,] state, Id, streak)
int VerticalStreak(row, column, [,] state, streak)
int HorizontalStreak(row, column, [,] state, streak)
int DiagonalCheck(row, column, [,] state, streak)



Dividimos las obligaciones tratando de aislar los comportamientos lo máximo posible, un GameManager para poder gestionar turnos y estado de la partida, un GameBoard para la creación, actualización y evaluación del tablero y una clase padre de jugadores para poder luego derivar jugadores humanos y jugadores automáticos donde implementar nuestros algoritmos de IA.

Llevó un tiempo ajustar el código del paquete a nuestra idea pero al final compensó el trabajo frente a las facilidades que nos otorgó posteriormente.

2.2. Función de evaluación

Inicialmente cometimos el error de despriorizar la función de evaluación, creíamos que dado el gran estudio del tema de inteligencias artificiales para jugar al cuatro en raya sería una simple cuestión de copiar y pegar código.

Cuando implementamos la primera versión de la función de evaluación nos dimos cuenta de que se trataba de un tema mucho más complicado del que inicialmente pensamos. Los resultados que devuelve eran inconsistentes y resultaban en una inteligencia artificial muy débil, pudiendo perder contra jugadores humanos en menos de 5 turnos consistentemente.

Gran parte del desarrollo de la práctica se perdió en intentar encontrar solución al problema.

Eventualmente tuvimos que reunirnos en grupo para afrontar conjuntamente el problema, tras varias modificaciones y cambios de funciones. Finalmente encontramos con una que podía proporcionar una inteligencia “Adecuada” a la ia, es capaz de interceptar los jugadas del jugador aunque por falta de tiempo no está todo lo realmente pulida que debería estar cometiendo errores de vez en cuando.

No es el resultado que esperábamos pero es una buena base.

2.3. Documentación de datos

Se realizó mediante un sistema que leía el juego por debajo, cogiendo estimaciones de nodos expandidos en la ejecución, tiempo medio que le llevó realizar los movimientos y la eficiencia de dividir ambos resultados anteriores.
Fue algo problemático y no se llegó a implementar una buena gestión visual por falta de tiempo, pero funciona correctamente en la consola de Unity a modo de Debug, en donde se pueden apreciar los resultados correctamente de las ejecuciones, aunque con los números de la columna de los tiempo medio algo más a la izquierda de lo que debería y casi juntándose a veces con los nodos expandidos.
Además, jugamos varias partidas y reunimos datos en un documento word anclado a esta práctica en el que se ven varios resultados de comprobaciones entre IAs.

3. Conclusiones

En base a las pruebas de más abajo, podemos llegar a la conclusión por eficiencia en términos generales es el aspiracional, ya que logra en la mayoría de casos una mayor eficiencia haciendo cálculos de los nodos expandidos entre el tiempo medio total de hacer movimientos en toda la partida.

Pruebas

Algorithm	***Expanded Nodes***	***Average Time***	***Effectiveness***
MINIMAX	475304	6560527	0,07244906
MINIMAX	437876	6043396	0,07245529
MINIMAX	490900	6749270	0,07273379
NEGAMAX	12984	274239,7	0,04734545
MINIMAX	632080	1,768191E+07	0,03574726
ASPIRATIONAL	71333	2377845	0,02999901
NEGAMAX	12192	253069,8	0,04817643
MINIMAX	432807	6060555	0,07141376
NEGAMAX	13348	313038,6	0,04264011
NEGAMAX	12439	326949,5	0,03804563
NEGAMAX	18970	661659,6	0,02867033
ASPIRATIONAL	82571	2933304	0,02814949
ASPIRATIONAL	57192	2892935	0,01976954
MINIMAX	492347	1,551617E+07	0,03173123
ASPIRATIONAL	87597	3673517	0,02384554
NEGAMAX	13124	455177,3	0,02883272

ASPIRATIONAL	45144	4753179	0,009497643
ASPIRATIONAL	35111	2892493	0,01213866