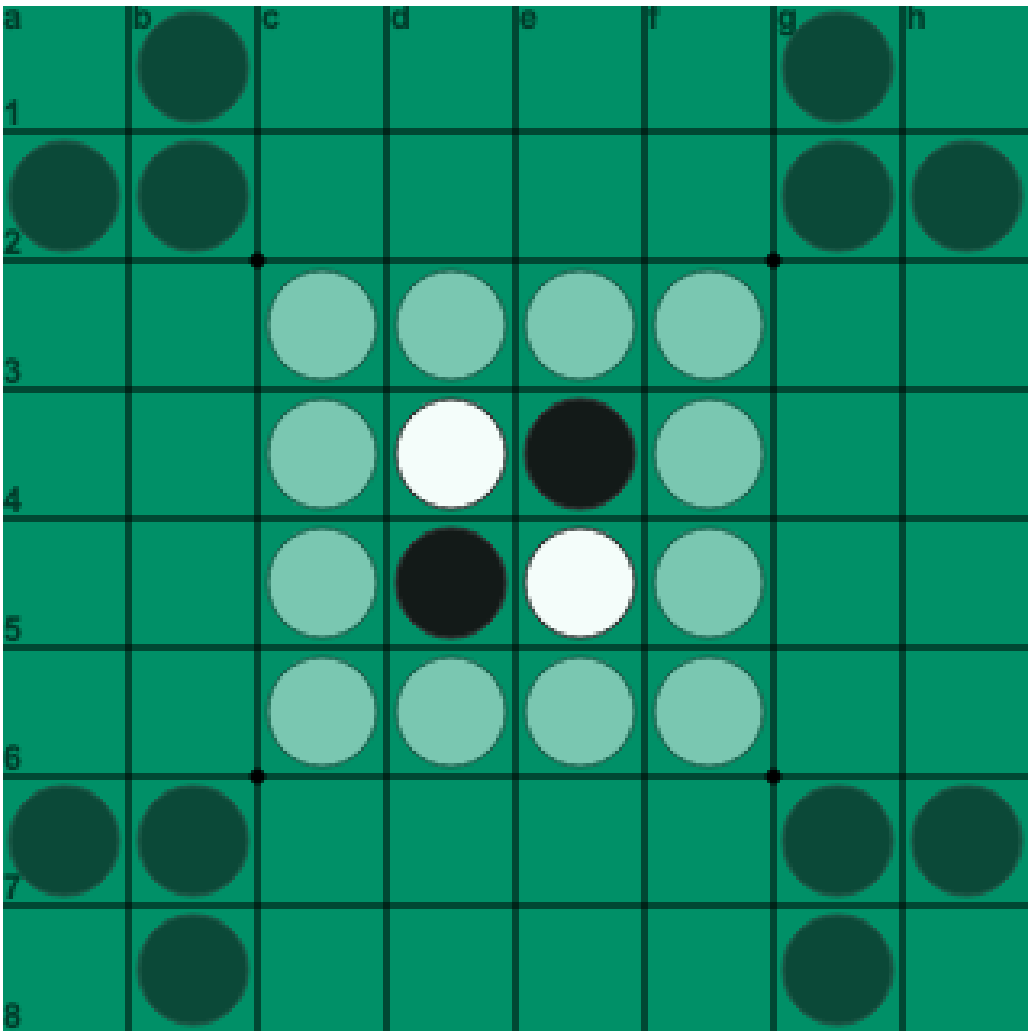
OTHELLO - Projecte Final PROP



OMatic

Ixent Cornella, Eric Gonzalez

GitHub Project

Index

1. Introducció

En aquest treball, es presenta la creació d'un jugador de Othello basat en l'algorisme minimax amb poda alfa-beta. L'algorisme avalua totes les possibles jugades que un jugador pot realitzar, així com les respostes de l'oponent, i selecciona la jugada que maximitza la puntuació del jugador, minimitzant al mateix temps la puntuació de l'oponent. La poda alfa-beta és una variant de l'algorisme minimax que s'utilitza per a accelerar el procés de cerca de la millor jugada, eliminant de l'avaluació aquelles branques de l'arbre de joc que no són prometedores.

S'han desenvoluparan dues versions del jugador de **Othello** basat en l'algorisme minimax amb poda alfa-beta:

PlayerMinimax: aquesta versió del jugador estarà parametritzada amb el nombre de nivells que s'exploren de forma fixa (la profunditat).

PlayerID: aquesta versió del jugador utilitzarà el mecanisme de timeout previst per la interfície gràfica, cosa que significa que el jugador tindrà un temps límit per a prendre una decisió. Per a poder tallar en qualsevol moment la cerca, s'implementarà l'algorisme Iterative-Deepening, que consisteix en anar augmentant progressivament el nivell de profunditat de la cerca fins que s'arribi el temps límit.

L'heurística és la part clau d'aquest treball, ja que determinarà l'eficiència i la capacitat del jugador per a prendre decisions estratègiques. Per tant, és fonamental pensar acuradament què tenir en compte i com calcular-la de forma eficient.

2. Heurística

En primer lloc, hi han diferents factors que es poden combinar per a obtenir una heurística altament eficient, com poden ser la paritat, la mobilitat, els corners, o l' estabilitat. Cal tenir en compte que aquests factors han de ser considerats com una combinació de tots ells, ja que cap dels factors serà el més adequat per totes les situacions, per exemple, si es contés només les peces del nostre color, seria un error, ja que la posició de les peces que triem és igual o més d'important que la quantitat.

Dit això, vam crear una classe per a l'heurística, i vàrem començar a experimentar amb diferents paràmetres per a la nostra heurística, i en total, es van recopilar tots aquests factors:

Factors

Paritat → es refereix a l'equilibri de peces blanques i negres en el tauler. Un jugador amb més peces del seu color en el tauler té avantatge, ja que disposa de més opcions per a realitzar moviments i voltejar les peces del seu oponent. No obstant això, no sempre és necessari prioritzar la paritat per sobre de tota la resta. Per exemple, si un jugador té l'opció de fer un moviment que augmenti la seva paritat però també li dona a l'oponent l'oportunitat de fer un moviment que augmenti la seva pròpia paritat, pot ser més beneficiós per al jugador fer un moviment diferent.

Pel que fa a la nostra implementació, la funció `hParitat()` retorna un enter que representa la puntuació de l'estat actual del joc. La puntuació es calcula tenint en compte la quantitat de les fitxes del jugador i de l'oponent en el tauler, i realitzant la resta entre el nombre de peces del jugador, i el nombre de peces de l'oponent. Cal afegir que el resultat obtingut es multiplicarà per un pes baix, com per exemple **1,** ja que la paritat no té tanta importància com, per exemple, capturar els corners.

En la següent situació, el jugador actual té 12 fitxes i l'oponent té 10 fitxes. La funció `hParitat()` retornaria una puntuació de **2** (ja que es faria 12 - 10 i després es multiplicaria per **1**).

Estabilitat → Es la capacitat de que una peça pugui cambiar de color o no. En la nostra pràctica només detectem l'estabilitat per les peces de les vores, per dos motius:

- Si haguéssim de calcular l'estabilitat del tauler, augmentariem el cost computacional del programa, ja que, per cada peça, s'hauria de comprovar diferents condicions.
- La resta de peçes depenen de les fitxes de les vores per a poder ser estables.

Per tant, si calculem si una vora és estable (només cal mirar si hi ha una seqüència de fitxes del mateix color fins a un corner), tenim una estabilitat també bastant forta.

Pel que fa a la nostra implementació, la funció `hEdges()` retorna un enter que representa el nombre de peçes a vores (edges) del taulell, i també les vores estables. La puntuació es calcula tenint en compte les fitxes del jugador i de l'oponent que es troben en les vores del tauler (les files i columnes 0 i 7). Després, s'utilitza una ponderació diferent per a les vores estables i les vores normals (la suma és inclusiva, de manera que una vora estable també es suma a les vores). S'utilitza la funció auxiliar `checkEdgeEstable(fila, columna, GameStatus)` per a comprovar si la vora es estable.

Corners → Quan ens referim als corners, ens referim a les posicions: (0,0), (0,7), (7,0) i (7,7), en un taulell 8x8. Aquestes posicions són especials perquè una vegada que són capturades per un jugador, no poden ser girades per l'oponent. A més, proporciona estabilitat a les peces del jugador en aquesta zona del tauler. Hi ha una alta correlació entre el nombre de cantonades capturades per un jugador i la seva probabilitat de guanyar la partida. Encara que capturar la majoria de les cantonades

no garanteix necessàriament la victòria, permet construir una major estabilitat en el tauler i per tant pot ser beneficiós per al jugador.

Pel que fa a la nostra implementació, la funció `hCorners()` retorna un enter que representa la puntuació de l'estat actual del joc. La puntuació es calcula tenint en compte les cantonades del tauler (les caselles en les posicions (0,0), (0,7), (7,0) i (7,7)). Si el jugador té una fitxa al corner, es suma a la variable *corners,* però si la té l'oponent, se li resta un a *corners*. Se li dona una ponderació bastant alta ja que considerem que es un factor altament important, i s'ha demostrat al càlcul de ponderacions.

Mobilitat → es refereix al nombre de moviments disponibles que té un jugador. Un jugador amb més mobilitat té més opcions per a fer moviments i voltejar les peces del seu oponent.

La funció `hMobilitat()` en primer lloc, obté el tamany de la llista de moviments, `s.getMoves().size()`, que realment és el número de moviments disponibles del jugador. El resultat obtingut el multipliquem pel seu pes corresponent.

Ponderacions

Ara calia donar ponderacions als diferents factors. Després de moltes proves, (figura següent), vam arribar a les següents ponderacions:

```
private static final int POND_CORNERS = 50;
private static final int POND_EDGES = 15;
private static final int POND_SEDGES = 30;
private static final double POND_MOBILITAT = 2;
private static final double POND_PARITAT = 2;
```

Tot i això, vam pensar que amb això no hi hauria suficient, i vam afegir una constant més que estableix un límit de jugades del jugador (cal pensar que les jugades reals son les de la variable *turns*, multiplicat per 2, ja que no contem les de l'oponent). Així podem variar la ponderació de Mobilitat i Paritat depenent de les jugades fetes.

```
private static final int TURN_TRESHOLD = 20;
```

Això és important ja que:

- Mobilitat és important de maximitzar al principi ja que es tindrà més llibertat de moviment, però cap al final ja no importa tant.
- La paritat no és important al principi, però és en el que es basa el joc per donar la victòria (qui té més peces guanya).

```
paritat = (turns > TURN_TRESHOLD) ?  paritat*2 : paritat;
mobilitat = (turns > TURN_TRESHOLD) ?  s.getMoves().size()*2 :
s.getMoves().size();
```

Condicions: 2 segons de timeout, 10 partides contra Desdemona

Mobilitat	Edges	Edges Estables	Corners	Paritat	WINRATE%
1	20	60	100	1	90,00%
2	20	60	100	2	80,00%
2	10	30	50	2	60,00%
2	15	60	100	2	80,00%
2	15	50	100	2	100,00%

3. PlayerMiniMax i PlayerID

3.1 MiniMax

En aquest jugador, només calia implementar l'algoritme MinMax amb una profunditat fixada per l'usuari. Per tant, s'ignorava la funció de timeout. D'aquesta manera, l'únic que hem fet és utilitzar el pseudocdi de les transparències de l'assignatura i adaptar-lo al Othello, però essencialment ens queden tres funcions importants: **MinMax** (que decideix el millor moviment), **MaxValor** (maximitza el valor del jugador i crida la funció minvalor), i **MinValor** (minimitza el valor del oponent). D'aquesta manera, simulem una "batalla" entre el jugador i l'oponent, i decidim tirar a on ens doni millors resultats.

El problema sorgeix quan la profunditat és un nombre alt (> 6). Això provoca un arbre de decisions (moviments fets) molt extens, fet que provocaria un alt cost temporal per la quantitat de calculacions a fer. Aquí entra en joc la reducció de nodes a calcular: Alfa-beta pruning.

La poda alfa beta es basa en el fet de que algunes "branques" de l'arbre són molt ineficients, i per tant, no té sentit seguir aquell camí. D'aquesta manera, mantindrem dues variables al minmax, **alfa** i **beta**, que ens serviran per detectar quan hi ha aquests valors ineficients, i deixar de cercar en aquella branca. Això redueix considerablement el nombre de nodes a buscar, millorant l'eficiència dràsticament.

3.2 Iterative Deeping (ID)

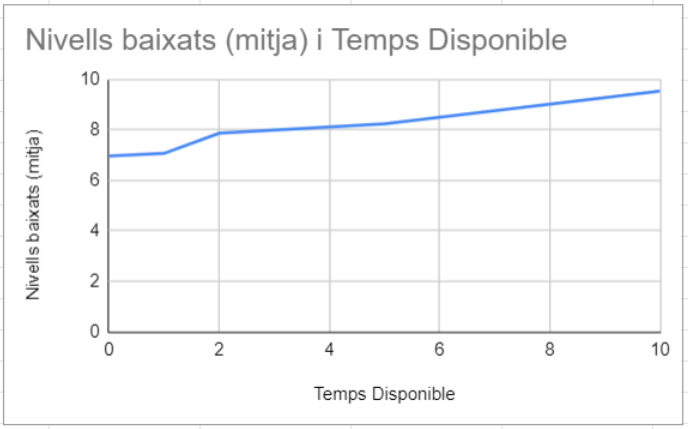
L'algorisme Iterative Deepening és una variant de l'algorisme Minimax. En lloc d'explorar tot l'arbre de joc d'una sola vegada, amb una profunditat fixada, l'algorisme Iterative Deepening explora l'arbre de incrementant la profunditat a cada iteració, fins que no es cridi el timeout. D'aquesta manera, garantim un bon moviment tot i que salti el timeout (a PlayerMiniMax, si tinguessim timeout, ens quedariem sense moviment si la cerca no ha acabat i ha saltat el timeout).

Per tant, aquest algoritme és en essència el mateix que PlayerMiniMax, però es repeteix, tota l'estona amb una profunditat incrementada fins a que salti el timeout.

4. Gràfica de número de nivells baixats

Aquesta gràfica mostra els nivells baixats en funció del timeout, per ilustrar la millora que provoca l'IDS:

Temps Disponibl	Nivells baixats (mitja)
0	6,967
1	7,067
2	7,871
5	8,241
10	9,542



5. Estratègies d'optimització

5.1 Poda alfa-beta

Aquesta poda l'hem comentat abans, i és la que guarda dos valors, alfa i beta, que ens serveixen per purgar l'arbre de cerca.

Al nostre codi s'utilitza de la següent manera:

```
valor = Math.max(valor, minValor(aux, alfa, beta, profunditat-1));
                if (beta < valor){
                    return valor;
                }
                alfa = Math.max(valor,alfa);
```

i també:

```
valor = Math.min(valor, maxValor(aux, alfa, beta, profunditat-1));
                if (valor < alfa){
                    return valor;
                }
                beta = Math.min(valor,beta);
```

Alfa i beta s'utilitzen per a detectar quan un valor d'heurística és realment dolent, així podrem deixar d'investigar aquella branca. Això sí, cal tenir una branca de referència, per tant podriem dir que fins que no s'ha explorat la primera branca (com a mínim) no podem fer una poda eficient.

Hem notat que sense la poda alfa beta, s'explorava una mitja de uns 2 milions de nodes, mentre que amb poda alfa beta s'exploren, com a molt, 600.000 mil nodes amb profunditat 8 i 10 moviments a explorar. Per tant, la millor és considerable, i la profunditat 8 és normalment sempre assolida amb IDS.

5.2 Poda Prob-Cut

Un altre tipus d'optimització podria ser la poda Prob-Cut, la qual és una tècnica de poda basada en donar una probabilitat a cada node, amb la fórmula **possibles_jugades ^ -profunditat**. D'aquesta manera, si la probabilitat obtinguda supera el límit donat, podem dir que no és óptima. Nosaltres hem fet una implementació d'aquesta poda, tot i que per falta de temps no hem pogut integrar del tot, i per tant, per defecte està desactivada però es pot activar al constructor de la classe (amb una flag). La probabilitat límit és de 0.1, cosa que està molt bé, ja que el branching factor de l'othello (moviments a realitzar) és de 10, per tant és molt probable que la probabilitat d'un node sigui més baixa que 0.1 (per la fórmula).

5.3 Zobrist Hashing i Taules de Transposició (no les hem utilitzat)

Pel que fa a un altre mètode d'optimització, podria ser el de implementar Taules de transposició amb l'ajuda de zobrist hashing (hem implementat una classe per la taula de transposicions, i dos funcions que es poden observar en el codi, per a inicialitzar i emplenar una taula amb zobrist hashing. Tot i això no ha acabat de funcionar correctament, i per tant, les hem borrat).

En primer lloc, la taula de transposició és una taula hash que s'utilitza per a emmagatzemar el resultat de l'avaluació d'una posició de l'arbre de joc. Quan s'arriba a una posició de l'arbre que ja ha estat avaluada prèviament, en lloc de tornar a avaluar-la es pot recuperar el resultat emmagatzemat en la taula de transposició i utilitzar-lo directament. D'aquesta manera, s'evita haver d'explorar de nou una part de l'arbre i s'estalvia temps de cerca.

El hashing de Zobrist és una tècnica que s'utilitza per a generar claus úniques per a cada posició de l'arbre de joc. Aquestes claus s'utilitzen per a emmagatzemar i recuperar els resultats de l'avaluació de les posicions en la taula de transposició. L'objectiu d'aquesta tècnica és generar claus úniques que siguin el més compactes possible i que siguin fàcils de calcular.

Othello - Eric i Ixent

```
    public static void ini_taula(){
        // Inicialitzem la taula amb numeros aleatoris
        for(int i = 0; i < 8; i++){
            for(int j = 0; j < 8; j++){
                for(int z = 0; z < 2; z++){
                    zTable[i][j][z] = Math.abs(rand.nextLong());// retorna un numero entre
0 i 9223372036854775807
                }
            }
        }
    }

    public static long zobristhash(GameStatus s){
        long hash = 0;

        for(int i = 0; i < 8; i++){
            for(int j = 0; j < 8; j++){
                // Si la fitxa es nostra, omplenem la posició 0
                if(s.getPos(i, j) == jugador){
                    hash ^= zTable[i][j][0];
                }
                // Si la fitxa es del oponent, omplenem la posició 1
                else if(s.getPos(i, j) == CellType.opposite(jugador)){
                    hash ^= zTable[i][j][1];
                }
            }
        }
        return hash;
    }
```

6. Repartició de les tasques

Per aquest treball, hem anat apuntant totes les reunions i tasques que cadascun de nosaltres ha hagut de realitzar. Seguidament es mostra una taula amb els resultats

Repartició tasques	
HEURISTICA	Ixent → 50% Eric → 50%
OPTIMITZACIÓ	Ixent → 60% Eric → 40%
DOCUMENTACIÓ	Ixent → 40% Eric → 60%

REUNIONS	
25/11/2022 (16:00 - 18:30)	Primera reunio per entendre el funcionament de la practica
28/11/2022 (16:00 - 21:00)	Identificació de possibles heuristiques
29/11/2022 (15:40 - 19:10)	Implementació primera heuristica simple
30/11/2022 (19:00 - 22:00)	Implementació primera heuristica simple
1/11/2022 (10:00 - 24:00)	Implementació heuristica final
3/11/2022 (11:00 - 14:15)	Implementació heuristica final
4/11/2022 (17:00 - 20:00)	Implementació heuristica final
9/12/2022 (11:00 - 12:00)	Implementació heuristica final
15/12/2022 (11:00 - 12:00)	Prob-Cut
17/12/2022 (16:00 - 19:00)	Prob-Cut
22/12/2022 (16:00 - 19:00)	Documentació
23/12/2022 (16:00 - 19:00)	Documentació