

## MINIMAX:

```
@Override
public int moviment(Tauler t, int color)
{
    this.nodes = 0;
    this.color_fitxes = color;
    int mov = moviment_rekursiva(t, color, profunditat: 0, ultimov: -1, alpha: Integer.MIN_VALUE, beta: Integer.MAX_VALUE);
    System.out.println(this.nodes + " nodes visitats.");
    return mov;
}
```

Per implementar minimax hem creat una funció moviment\_rekursiva, cridada des de moviment i que, com el seu nom indica, és recursiva. Li passem com a paràmetres el tauler, el nostre color, iniciem amb profunditat 0, últim moviment és -1 perquè no s'ha fet cap moviment encara, i els valors alpha i beta al mínim i màxim possibles respectivament.

```
public int moviment_rekursiva(Tauler t, int color, int profunditat, int ultimov, int alpha, int beta){
    int millor_columna = 0;
    if (ultimov >= 0 && (profunditat >= nivell_max || !t.espotmoure() || t.solucio(c: ultimov, clr: color))){
        return heuristica(t, ultimov, color*-1);
    }
    else {
```

El primer que fem és comprovar si es tracta d'una fulla, és a dir, si s'ha arribat a la profunditat que li hem dit, si el tauler està ple, o si algú ha guanyat. En aquest cas es retorna la heurística.

```
    else {
        int heuristica = 0;
        for (int i = 0; i < t.getMida(); ++i){
            if (t.movpossible(col: i)){
                if (!alphaBeta || (alphaBeta && beta > alpha)){
                    Tauler t2 = new Tauler(t);
                    t2.afegeix(columna: i, color);
                    if (profunditat % 2 == 0) {
                        heuristica = moviment_rekursiva(t: t2, color*-1, profunditat+1, ultimov: i, alpha, beta);
                        if (heuristica > alpha) {
                            alpha = heuristica;
                            millor_columna = i;
                        }
                    }
                    else {
                        heuristica = moviment_rekursiva(t: t2, color*-1, profunditat+1, ultimov: i, alpha, beta);
                        if (beta > heuristica){
                            beta = heuristica;
                        }
                    }
                }
            }
        }
        if (ultimov >= 0){
            if (profunditat % 2 == 0) return alpha;
            else return beta;
        }
    }
    if (ultimov == -1) System.out.println("heuristica: " + alpha);
    return millor_columna;
```

Sinó, llavors recorrem les possibles tirades, és a dir, cada columna que no estigui plena, fem una còpia del tauler i li afegim la fitxa. Amb el nou tauler cridem recursivament la funció `moviment_recurativa`, indicat a `ultmov` a quina columna hem col·locat la fitxa, augmentant la profunditat i canviant el color. Quan arribem el cas en el que els fills d'un node siguin fulles, llavors anirem actualitzant els valors alpha o beta per mantenir el valor màxim o el mínim. Detectem si estem a un nivell de l'arbre Max o Min mirant si la profunditat és parella o no, ja que com sempre comencem amb Max a profunditat 0, sabem que les profunditats parelles pertanyen als nivells max, i la resta al min. Quan trobem una columna amb un valor millor per a alpha, la col·loquem com a millor columna. Un cop s'han acabat de mirar totes les heurístiques, la funció retornarà amb la heurística que s'ha quedat, i el node pare al que li arribarà el valor farà el mateix procés, fins arribar a l'arrel, que diferenciarem perquè `ultmov` (últim moviment) val -1, com hem indicat abans, no retornarà cap heurística sinó que retornarà la millor columna, que serà a la que col·locarem finalment la fitxa en el joc.

#### Poda alpha-beta:

Com es veu en el codi, abans de crear un fill amb la tirada a la columna `i`, mirem si `alphaBeta` està activat, i si ho està, només entrem si beta és major que alpha, de manera que podem les branques tals que  $\alpha \geq \beta$ .

El resultat és el mateix que si no podem, ja que només deixem de visitar branques que no tenen bones solucions, però el nombre de nòds explorats disminueix considerablement, fent que sigui més ràpid.

Aquest és un exemple amb profunditat 4, sense poda:

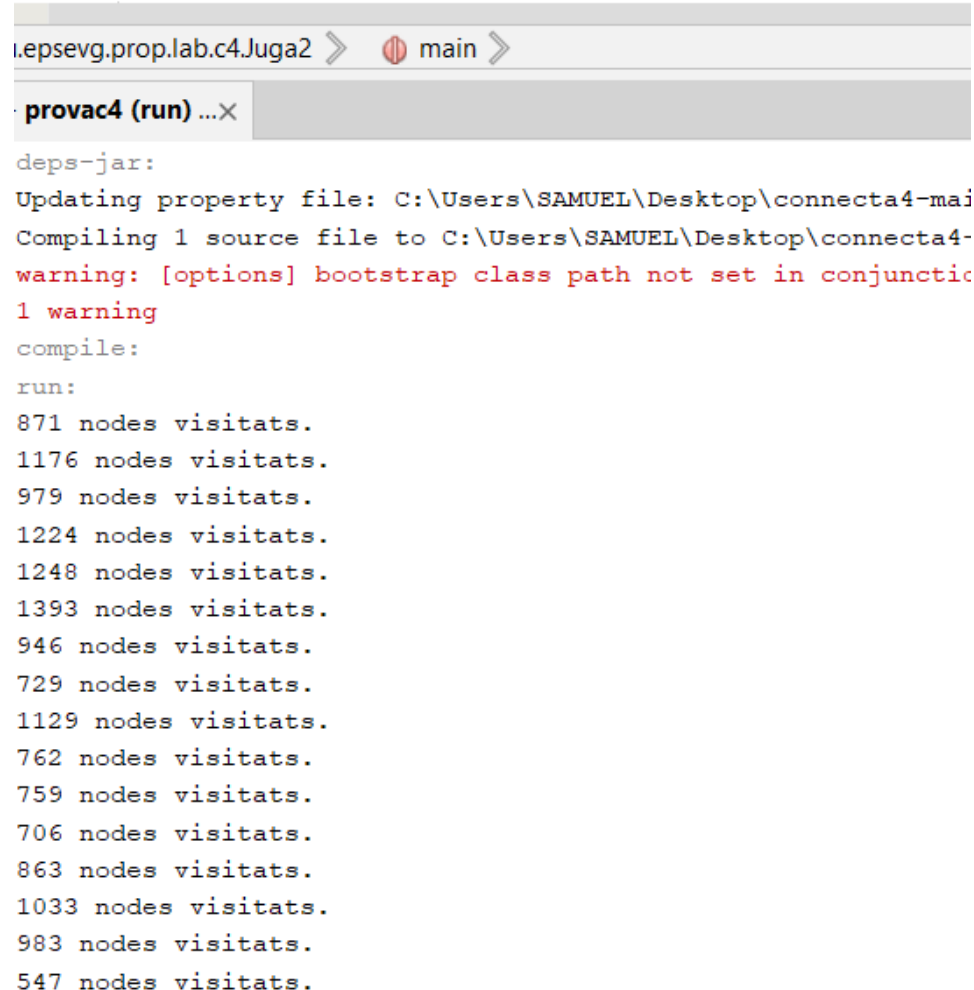
```
Jugador p1 = new C4(nivell:4, poda: false);
u.epsevg.prop.lab.c4.Juga2 > main > p1 >
- provac4 (run) ...X
deps-jar:
Updating property file: C:\Users\SAMUEL\Desktop\connecta4-mai
Compiling 1 source file to C:\Users\SAMUEL\Desktop\connecta4-
warning: [options] bootstrap class path not set in conjunctio
1 warning
compile:
run:
4096 nodes visitats.
4096 nodes visitats.
4096 nodes visitats.
4012 nodes visitats.
4012 nodes visitats.
3669 nodes visitats.
4045 nodes visitats.
4092 nodes visitats.
4041 nodes visitats.
4070 nodes visitats.
3925 nodes visitats.
3874 nodes visitats.
3763 nodes visitats.
3720 nodes visitats.
3405 nodes visitats.
3076 nodes visitats.
```

Com es pot veure, comença visitant 4096 nodes, resultat lògic, ja que és 8 elevat a 4, visitem tots els nodes fulla que hi ha a profunditat 4, tenint en compte que cada node té m fills, sent m el número de columnes, el que ens queda es que es visiten  $m^n$  nodes on n és la profunditat a la que baixem.

Naturalment segon anem omplint el tauler, el nombre de nodes visitats disminueix, ja que es troben els fills a menor profunditat, i no tots els nodes tenen 8 fills, ja que és més probable que hi hagi columnes plenes.

Fem la comparació amb profunditat 4 però amb poda:

```
Jugador p1 = new C4(nivell:4, poda: true);
```



```
deps-jar:
Updating property file: C:\Users\SAMUEL\Desktop\connecta4-mai
Compiling 1 source file to C:\Users\SAMUEL\Desktop\connecta4-
warning: [options] bootstrap class path not set in conjunction
1 warning
compile:
run:
871 nodes visitats.
1176 nodes visitats.
979 nodes visitats.
1224 nodes visitats.
1248 nodes visitats.
1393 nodes visitats.
946 nodes visitats.
729 nodes visitats.
1129 nodes visitats.
762 nodes visitats.
759 nodes visitats.
706 nodes visitats.
863 nodes visitats.
1033 nodes visitats.
983 nodes visitats.
547 nodes visitats.
```

La quantitat de nodes visitats és, aproximadament, 4 vegades menor que sense.

Fins ara hem estant mirant les columnes d'esquerra a dreta, si provem amb ordre invers, començant en la columna 7 i acabant en la 0, no tan sols podarem diferent i el nombre de nodes visitats serà diferent, sinó que a més el resultat i les jugades escollides seran unes altres;

```
//Jugador p1 = new Aleatori();
Jugador p1 = new C4(nivell:4, poda:true);

.epsevg.prop.lab.c4.Juga2 >>

provac4 (run) ...X

Deleting: C:\Users\SAMUEL\Desktop\connecta4-main\c4_the_
deps-jar:
Updating property file: C:\Users\SAMUEL\Desktop\connecta
Compiling 1 source file to C:\Users\SAMUEL\Desktop\conne
warning: [options] bootstrap class path not set in conjun
1 warning
compile:
run:
888 nodes visitats.
964 nodes visitats.
389 nodes visitats.
358 nodes visitats.
297 nodes visitats.
334 nodes visitats.
386 nodes visitats.
454 nodes visitats.
1152 nodes visitats.
446 nodes visitats.
383 nodes visitats.
316 nodes visitats.
1136 nodes visitats.
642 nodes visitats.
293 nodes visitats.
```

Fent un cap d'ull per sobre, veurem que els nodes visitats són lleugerament inferiors ara, però depèn de cada cas particular, en general començar per una banda o una altra no ens hauria d'afectar al rendiment.

## HEURÍSTICA:

La nostra heurística comença amb la detecció de solució, i retornem el mínim valor possible +1 si hem vist que perdem, i el màxim valor possible - 1 si guanyem.

```
public int heuristica(Tauler t, int ultmov, int colr){
    int h = 0;
    if (t.solucio(c: ultmov, clr: colr)){
        if (colr != color_fitxes) h = Integer.MIN_VALUE+1;
        else h = Integer.MAX_VALUE-1;
    }
    else {
```

Sinó, tenim el següent:

```

else {
    //COLUMNS
    for (int col = 0; col < t.getMida(); ++col){
        int ratxa = 0;
        int fila = t.getMida()-1;
        int color = t.getColor(f: fila, c: col);
        int buides = 0;
        while (color == 0 && fila > 0){
            ++buides;
            --fila;
            color = t.getColor(f: fila, c: col);
        }
        if (color != 0){
            while (fila >= 0 && t.getColor(f: fila, c: col) == color){
                if (color == color_fitxes) ++ratxa;
                else --ratxa;
                --fila;
            }
        }
        else ++buides;
        if (buides + Math.abs(a: ratxa) >= 4){
            if (ratxa > 0) h += Math.pow(a: ratxa, b: ratxa);
            else if (ratxa < 0) h -= Math.pow(a: Math.abs(a: ratxa), b: Math.abs(a: ratxa));
        }
    }
}

```

Primerament, recorrem les columnes. Comptem el nombre de caselles buides que hi ha, i un cop arribem a la fitxa més alta, comptem quantes hi han del mateix color seguides. Si al final veiem que hi ha opció de fer 4, és a dir, que entre les fitxes ja col·locades i les caselles buides, sumen 4 o més. Després sumem el valor de les fitxes ja col·locades elevat a ell mateix, si són fitxes del jugador, o li restem en cas contrari. D'aquesta manera avaluem la nostra situació en vertical, ja que només tenim en compte els llocs on encara es poden col·locar 4 fitxes iguals i donem més pes com més a prop de 4 estigui.

Per a les files seguim una lògica semblant, recorrem les files de baix a dalt, i les caselles d'esquerra a dreta fins arribar a la casella mida del tauler -4. El que fem és avaluar en grups de 4 caselles, el primer grup seria 0,1,2,3, i l'últim, en un tauler de 8x8, 4,5,6,7, per això només recorrem fins a la columna 4. Per cada grup de 4 mirem quantes fitxes de cada jugador n'hi ha, quants espais buits, quantes van seguides, i en cas de no ser a la fila més baixa, mirar si un espai buit té una fitxa a sota o no. Després sumem o restem el número de fitxes elevat a ell mateix i li sumem 2 pel número de fitxes consecutives màxim detectat, només quan hem determinat que és possible fer 4 en ralla en aquest grup de caselles. Diem que és possible fer 4 en ralla quan, només hi ha fitxes d'un jugador, no estan buides les 4 caselles, i en files superiors a la primera, als espais buits hi ha una fitxa a sota i per tant es pot col·locar una fitxa en aquella posició.

```
//FILES
for (int fila = 0; fila < t.getMida(); ++fila){
    boolean buida = false;
    for (int col = 0; col < t.getMida() - 3 && !buida; ++col){
        int buides = 0;
        int fitxes_jugador = 0;
        int fitxes_rival = 0;
        boolean possible4 = true;
        int consecutives = 0;
        int color = 0;
        int ratxa = 0;
        for (int i = 0; i < 4 && possible4; ++i){
            if (consecutives == 0 && t.getColor(f: fila, col+i) != 0) consecutives = 1;
            else if (color == t.getColor(f: fila, col+i)) ++consecutives;
            if (t.getColor(f: fila, col+i) == color_fitxes) fitxes_jugador++;
            else if (t.getColor(f: fila, col+i) == 0){
                ratxa = consecutives;
                buides++;
                if (fila > 0 && t.getColor(fila-1, col+i) == 0) possible4 = false;
            }
            else fitxes_rival++;
            if (fitxes_jugador > 0 && fitxes_rival > 0) possible4 = false;
            color = t.getColor(f: fila, col+i);
        }
        if (possible4){
            ratxa = Math.max(a: ratxa, b: consecutives);
            if (fitxes_jugador > 0) h += Math.pow(a: fitxes_jugador, b: fitxes_jugador) + 2*ratxa;
            else if (fitxes_rival > 0) h -= Math.pow(a: fitxes_rival, b: fitxes_rival)+2*ratxa;
        }
        if (buides == 4) buida = true;
    }
}
}
```

Encara que el rendiment no és dolent, creiem que és millorable. El que hem volgut però no ens hem ensortit era analitzar també les diagonals, però no ho hem aconseguit implementar de forma que millorés la heurística. Al final els nostres resultats són:

Com a Player1:

Profunditat 4: Guanya

Profunditat 6: Perd

Profunditat 8: Perd

Com a Player2:

Profunditat 4: Guanya

Profunditat 6: Perd

Profunditat 8: Taules