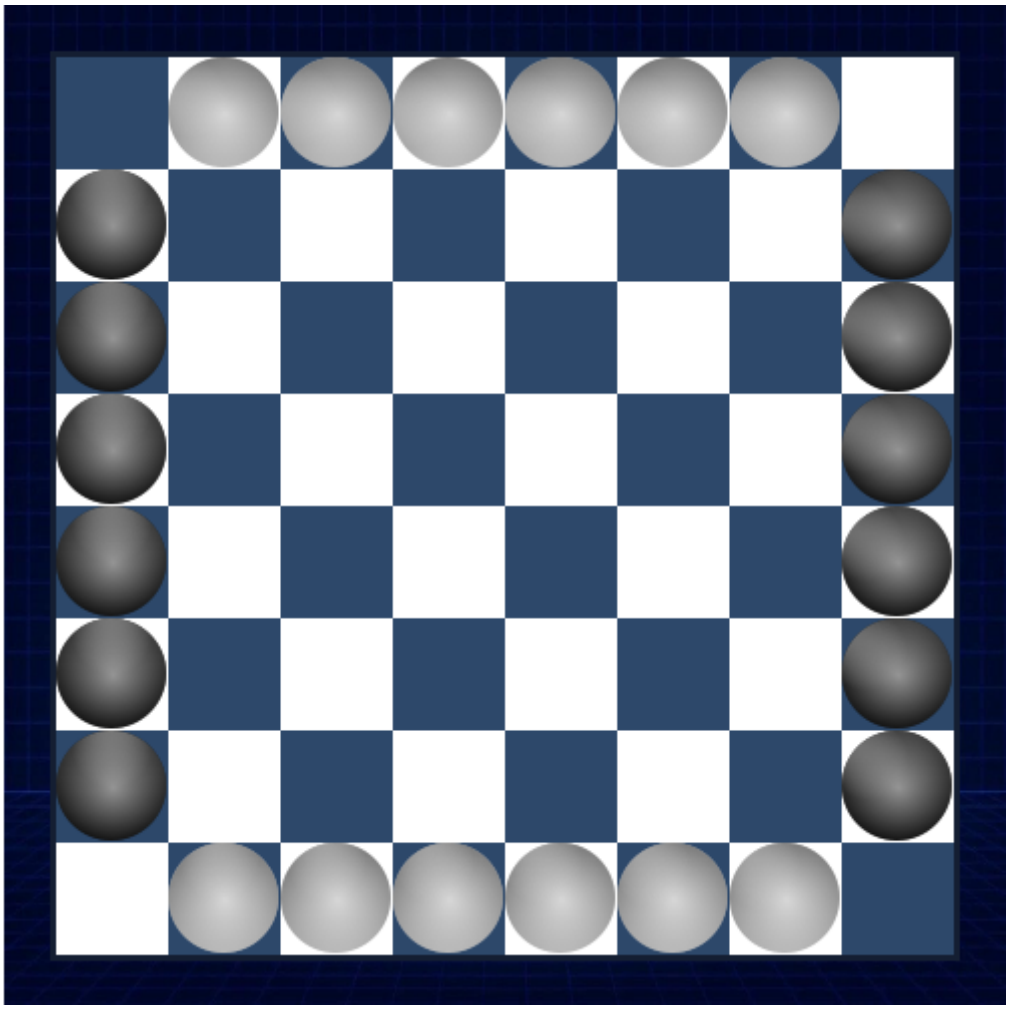
UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Projecte de Programació

Tematé Otravé

Documentació de la I.A. de Lines Of Action



EPSEVG - Q5
Eduardo Pinto
Oriol Fernández

Index

1. Heurístiques

A l'intentar crear una bona heurística capaç de vencer a `MCCloudPlayer`, vam intentar diverses estratègies que no donaven bons resultats. Però, això era perquè teniem un error a l'algoritme min-max, un cop solucionat, ens vam adonar compte de que les heuristiques creades no eres dolentes i vam decidir mantenir-les totes per no desaprofitar hores de treball i donar a la nostra IA l'opció de seleccionar dificultat. Llavors hem creat l'opció de triar contra quina heurística et vols enfrentar.

"La meva mare sempre deia que no s'ha de desaprofitar res"

Estudiant de PROP 2021

Per sel·leccionar l'heurística que volem utilitzar, farem servir un paràmetre extra al constructor del jugador `temate_otrave`.

```
public temate_otrave(String name, int profunditatMaxima, int profunditatInicial,
HeuristicaEnum heuristicaSeleccionada, MinmaxEnum minmaxSeleccionat) {
	this.name = name;
	this.profunditatMaxima = profunditatMaxima;
	this.profunditatInicial = profunditatInicial;
	this.heuristicaSeleccionada = heuristicaSeleccionada;
	this.minmaxSeleccionat = minmaxSeleccionat;
	...
}
```

A l'hora de calcular el valor de heurística, donarem valors a les fitxes dels dos jugadors i restarem la puntuació del jugador 2 respecte al jugador 1.
Per saber quina heurística utilitzar, fem servir un `switch case`.

```
public static int calcula(HeuristicaEnum heuristicaSeleccionada, ElMeuStatus estat,
CellType jugador) {
	switch (heuristicaSeleccionada) {
		case HEURISTICA_1: return heuristica_1(estat, jugador) -
	heuristica_1(estat, CellType.opposite(jugador));
		case HEURISTICA_2: return heuristica_2(estat, jugador) -
	heuristica_2(estat, CellType.opposite(jugador));
		case HEURISTICA_3: return heuristica_3(estat, jugador) -
	heuristica_3(estat, CellType.opposite(jugador));
		case HEURISTICA_4: return heuristica_4(estat, jugador) -
	heuristica_4(estat, CellType.opposite(jugador));
	};
	return 0;
}
```

1.1. Heurística_1

Per fer aquesta heurística, hem decidit puntuar segons el millor grup format i la seva posició dintre del tauler. Cridem al mètode `search_best_group`, que buscará el millor grup donat un conjunt de fitxes.

```
for (int i = 0 ; i < s.getNumberOfPiecesPerColor(jugador) ; i++) {
      fitxes.add(s.getPiece(jugador, i));
}
score = search_best_group(fitxes);
```

Mètode search_best_group

En aquest mètode, creem un array anomenat `visitades` que utilitzarem per evitar visitar fitxes que ja hem visitat anteriorment i evitar cicles.

```
ArrayList<Boolean> visitades = new ArrayList<>();
for (int i = 0; i < fitxes.size(); i++) {
      visitades.add(false);
}
```

Per cada fitxa de l'array, cridem al mètode `suma_veines` per donar un valor al conjunt que forma amb les altres fitxes de l'array, `suma_veines` s'encarrega de comprovar si les altres fitxes son veïnes i de donar valor al conjunt. I com que només volem el valor del millor grup, fem servir el `Math.max`.

```
int score = 1;
for (int i = 0; i < fitxes.size(); i++) {
       if(!visitades.get(i)) {
              score = Math.max(grup_max, suma_veines(i, fitxes, visitades));
      }

      if(score >= (fitxes.size() - i))
              break;
       visitades.set(i,true);
}
return score;
```

Mètode suma_veines

Aquest mètode, donat una fitxa, li dóna valor segons la seva posició(observeu la *Figura 1.1.1.)* i multiplica aquest valor per el número de veines que té, d'aquesta forma es dóna preferència a formar grups en comptes d'únicament anar cap al centre. Formar grups al centre dóna molts punts.

```
Point p = fitxes.get(posFitxa);
int valor = puntuacions[p.x][p.y] * num_veines(posFitxa, fitxes);
```

El següent que fem, és comparar si la fitxa actual, es veïna amb les altres fitxes de l'array, i en cas de que hi hagi un altre veïna, cridem de forma recursiva `suma_veines` marcant com visitades les fitxes que ja hem sumat, perquè si no, es formaria un cicle. En el for, començem per `i = posFitxa` perque sabem que les posicions anterios a `posFitxa` ja han estat visitades abans.

```
for (int i = posFitxa + 1; i < fitxes.size(); i++) {
       if ( (esVeina(p,fitxes.get(i))) && (!visitades.get(i)) ) {
              visitades.set(i,true);
            valor += suma_veines(posFitxa + 1, fitxes, visitades);
       }
}
```

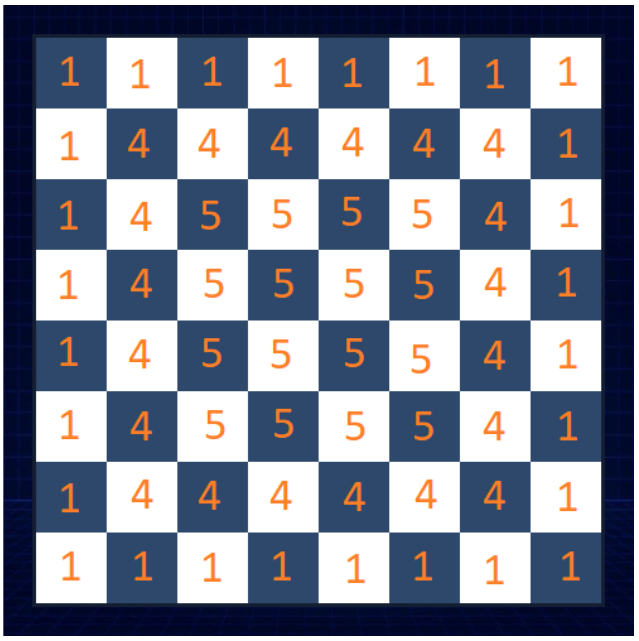


Figura 1.1.1. Puntuació segons la posició.

Hem decidit puntuar la posició de les fitxes segons el tauler de la *Figura 1.1.,* donant millor puntuació a les caselles centrals per assegurar-nos agafar una bona posició dintre del tauler.

Mètode num_veines

Aquest mètode és bastant sencill, únicament compte el número de fitxes veïnes del mateix color. Retornant el número de veïnes més 1(que sería ella mateixa).

```
public static int num_veines(int posFitxa, ArrayList<Point> fitxes) {
	Point p = fitxes.get(posFitxa);
	int valor = 1;

	for (int i = 0; i < fitxes.size(); i++) {
		if ( (esVeina(p,fitxes.get(i))) && (fitxes.get(i) != p) ) {
		    valor++;
		}
	}
	return valor;
}
```

La *Figura 1.1.2.* ens dóna un exemple de com es puntuaria aquest tauler. El numero que hi ha al centre de cada fitxa es la puntuació de la fitxa respecte el tauler, i el número més petit és el número de fitxes veïnes(comptant-se a si mateixa). Per cada grup de fitxes, es multiplicaria el número de cada fitxa per el número de veïnes i es sumaria a la puntuació del grup. En aquest cas, el millor grup sería el grup que está encerclat, donant una puntuació de 35 punts(5*2 + 5*3 + 5*2).

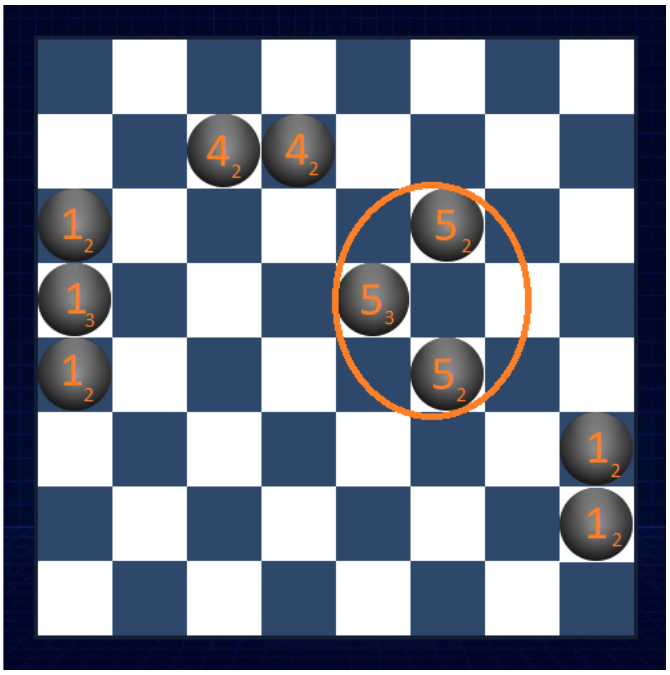


Figura 1.1.2. Tauler d'exemple.

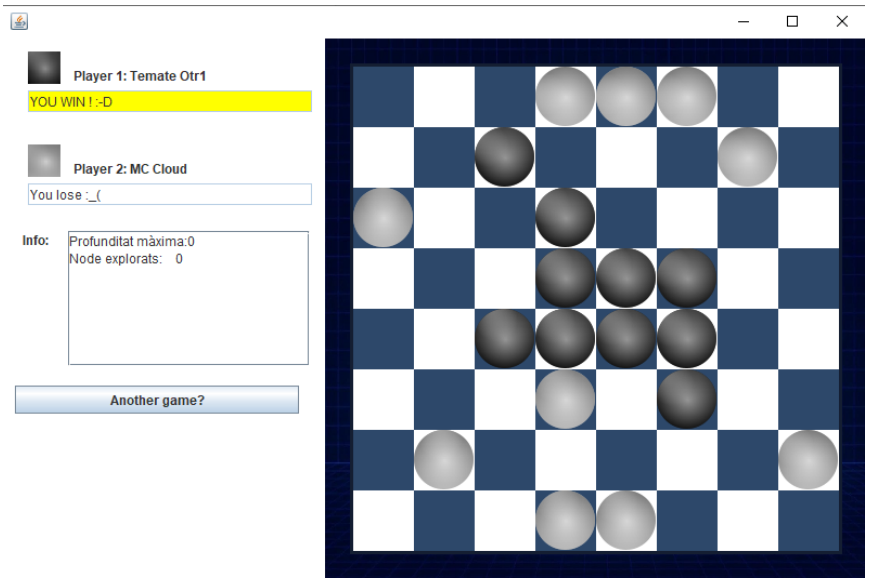


Figura 1.1.3. Victoria de l'heuristica_1 contra MC Cloud.

1.2. Heurística_2

En aquesta heurística fem servir només dos mètodes, `create_groups` per obtenir tots els diferents grups que tenim formats al tauler i `distancies` que dóna valor a heurística segons la distància que hi han entre les fitxes respecte als grups.
`fitxes` es un array amb totes les fitxes del mateix color.

```
for (int i = 0 ; i < s.getNumberOfPiecesPerColor(jugador) ; i++) {
	fitxes.add(s.getPiece(jugador, i));
}

groups = create_groups(s, jugador, fitxes);
score = distancies(groups, fitxes);

return score;
```

Mètode create_groups

Aquest mètode serveix per identificar els diferents grups formats amb les fitxes del mateix color.

Recorrem tots els grup per comprovar si la fitxa que estem visitant es veïna d'alguna fitxa d'algun grup, en cas afirmatiu, la fitxa ha de pertànyer a aquell grup.

```
for (ArrayList<Point> grup : grups) {
	for (Point p : grup) {
		if (esVeina(fitxes.get(i), p)) {
			grup.add(fitxes.get(i));
			afegida = true;
```

```
                }
                if (afegida) break;
        }
        if (afegida) break;
}
```

En cas que no pertany a cap grup, creem un grup i afegim la fitxa a aquell grup.

```
if (!afegida) {
        ArrayList<Point> g = new ArrayList<>();
        g.add(fitxes.get(i));
        grups.add(g);
}
```

La *Figura 1.2.1.* és un exemple de com es consideren els grups segons aquest mètode.

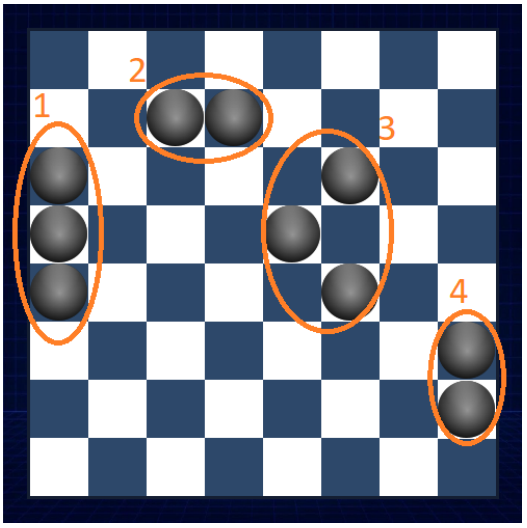


Figura 1.2.1. Tauler amb quatre grups formats.

Mètode distancies

Aquest mètode calcula la distància que hi ha entre els grups i les fitxes que no pertanyen al grup, i retorna la puntuació del grup que té les fitxes a menor distancia.

Comença comprovant quines `fitxes` no pertanyen al `grup`.

```
int score = Integer.MIN_VALUE;
for (ArrayList<Point> grup : grups) {
        int puntuacio = 0;
        for (Point p : fitxes) {
                if (!grup.contains(p)) {
                        ...
                }
        }
```

```
      if (puntuacio > score)
            score = puntuacio;
}
return score;
```

Si la fitxa no pertany al grup, comprova a quina distància està amb les altres fitxes del grup i es queda amb la distància més petita. I la distancia es resta a la `puntuacio`, per tant, el grup que tingui les fitxes més properes, tindrà la millorar `puntuacio`.

```
int min = Integer.MAX_VALUE;
for (Point p2 : grup) {
      int d = distance(p,p2);

      if (d < min)
            min = d;
}
puntuacio -= min;
```

Mètode distance

Calcula la distància entre dues fitxes.

```
private static int distance(Point a, Point b) {
      int x =  Math.abs(b.x - a.x);
      int y =  Math.abs(b.y - a.y);
      return Math.max(x, y);
}
```

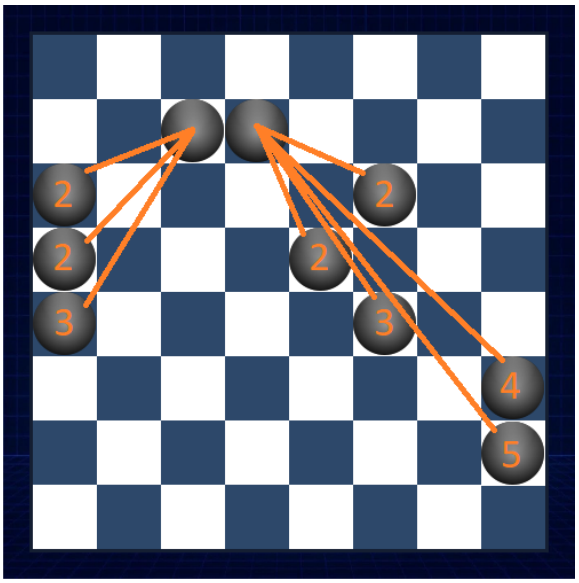


Figura 1.2.2. Distàncies entre el grup 2 i les altres fitxes

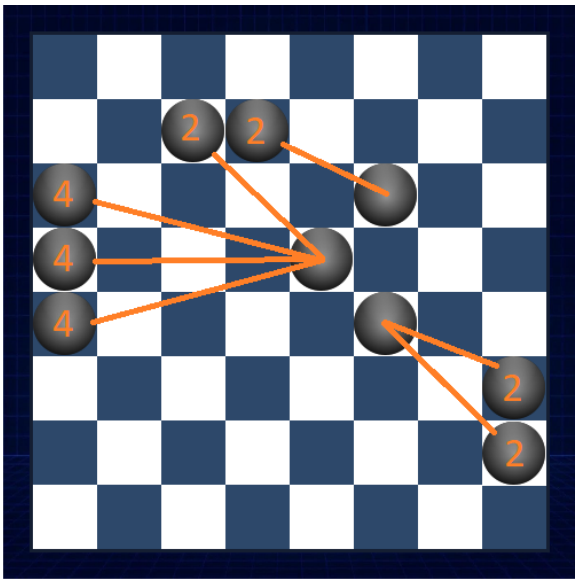


Figura 1.2.3. Distàncies entre el grup 3 i les altres fitxes.

Per tenir una idea de com puntua aquesta heurística, hem fet uns exemples a les *Figures 1.2.2* i *1.2.3.* La *Figura 1.2.2.* tindria un valor de -23 punts, mentres que la *Figura 1.2.3.* de -20. Llavors el grup 3 es qui té millor puntuació. I gràcies a aquesta heurística, farem que les altres fitxes s'apropin al millor grup, formant un únic grup i assegurant la victoria.

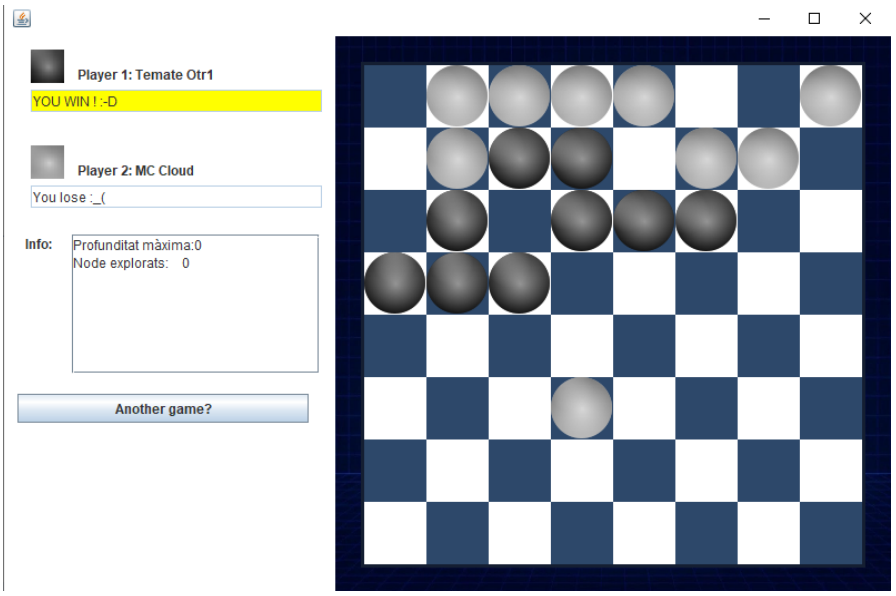


Figura 1.2.4. Victoria de l'heuristica_2 contra MC Cloud.

.

1.3. Heurística_3

Aquesta heurística s'assemble a la heurística_1, ja que dóna valor a cada casella del tauler, amb la diferencia de que en comptes de sumar punts, resta. Fa servir aquest dos métodes.

```
score += center_distances(fitxes);
score += puntua_veines(fitxes);
```

Mètode center_distances

Segons la posició de la fitxa, retorna un valor que s'afegeix a l'`score`.

```
int score = 0;
for(Point p: fitxes) {
      score += mat_score[p.x][p.y];
}
return score;
```

S'ha decidit puntuar segons la *Figura 1.3.1.*, donant valor de -50 a les esquines perquè són més fàcils de bloquejar, i com més llunyà del centre, més punts es perden, d'aquesta manera es prioritza anar cap al centre, tenint en compte, que si mengem una fitxa al rival, guanyarà punts. Hem fet això, perquè és més fàcil guanyar amb poques fitxes, llavors penalitzem d'aquesta forma menjar fitxes innecessàries.

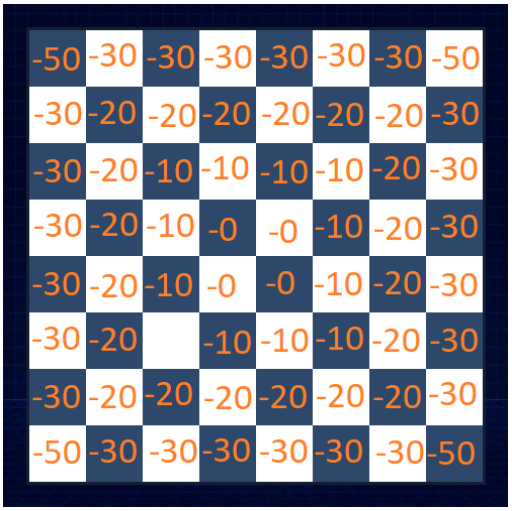


Figura 1.3.1. Puntuació segons la posició.

Mètode puntua_veines

Per cada fitxa de l'array `fitxes`, comprova amb totes les altres fitxes de l'array si són veïnes i com estan situades.

```
public static int puntua_veines(ArrayList<Point> fitxes) {
	int score = 0;
	int it = 1;
	for(Point p: fitxes) {
		for(int i = it; i < fitxes.size(); i++) {
			score += puntua_veina(p, fitxes.get(i));
		}
		it++;
	}
	return score;
}
```

Mètode puntua_veina

Comprovem si la seva veïna está en diagonal o recta, dependent de com sigui, varia la puntuació.

```
public static int puntua_veina(Point a, Point b) {
	int score = 0;
	int x = Math.abs(a.x - b.x);
	int y = Math.abs(a.y - b.y);
	if(x <= 0 && y <= 0) {
```

```
                if(x == 0 || y == 0)
                        score = 5; //recta
                else
                        score = 15; //diagonal
        }
        return score;
}
```

Hem decidit donar més puntuació a les diagonals, perquè les diagonal tenen més caselles veïnes, per tant, és més fàcil apropar una fitxa a aquestes caselles, en canvi, les rectes, tenen menys caselles veïnes, per tant, hem decidit donar-li menys puntuació.

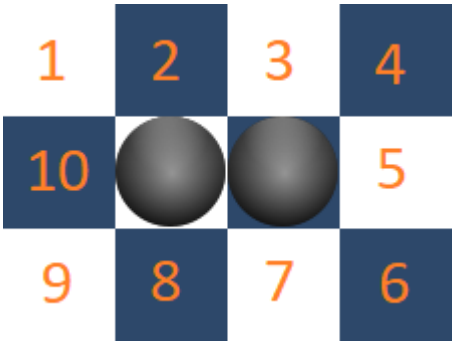


Figura 1.3.2. Puntuació de 5.

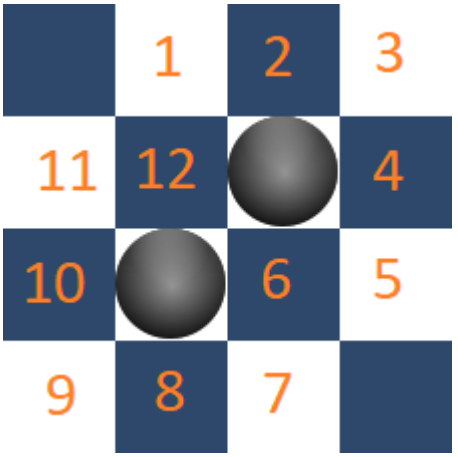


Figura 1.3.3. Puntuació de 15.

En aquest cas, el tauler es puntuaría de la següent manera:

- Per cada fitxa, se li resta punts segons la seva ubicació.
- Segons les veïnes que tingui, se li sumen punts.

La *Figura 1.3.4.* obtendría un valor de -160. Es normal tenir valors negatius amb aquesta heurística, però sempre agafarem la millor jugada, que serà la que tingui el valor menys negatiu.

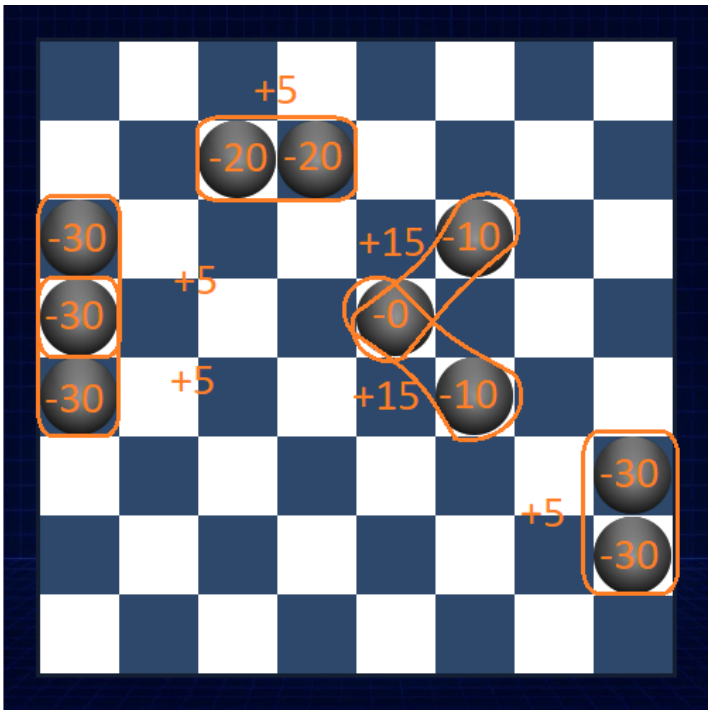


Figura 1.3.4. Tauler d'exemple.

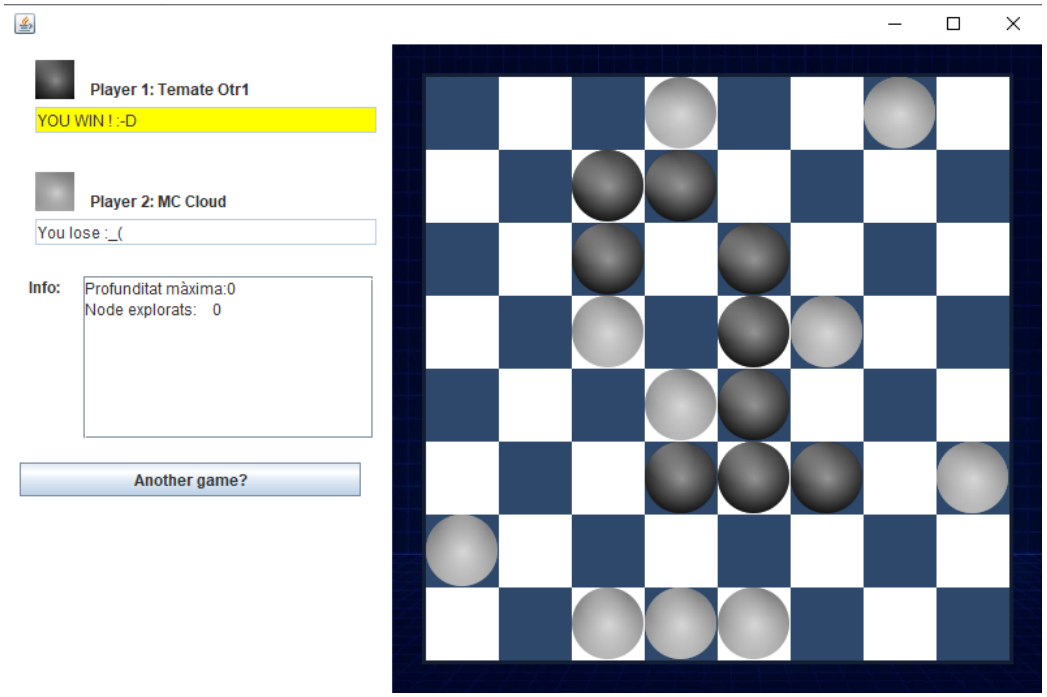


Figura 1.3.5. *Victoria de l'heuristica_3 contra MC Cloud.*

1.4. Heurística_4

Aquesta heurística és semblant a l'heurística_3, amb la diferència de que es triarà una casella del tauler que sigui el centre de masses, i restarà punts segons la distància de les fitxes amb el centre de masses.

```
Point p = get_center_of_mass(fitxes);
score += distances_from_center(p, fitxes);
score += puntua_veines(fitxes);
```

També fa servir el mètode `puntua_veines` explicat anteriorment per donar puntuació als grups formats.

Mètode get_center_of_mass

Donat un array de les `fitxes`, calcularà la mitjana de les posicions i retornarà la casella que se suposa que està al centre de totes les fitxes.

```
public static Point get_center_of_mass(ArrayList<Point> fitxes) {
	double x = 0, y = 0;
	for(Point p: fitxes) {
		x += p.x;
		y += p.y;
	}
	x = Math.round(x/fitxes.size());
	y = Math.round(y/fitxes.size());
	return new Point((int)x, (int)y);
}
```

A la *Figura 1.4.1.* podem observar quin sería el centre de masses. L'estratègia sería reduir la distància de les fitxes amb el centre de masses.

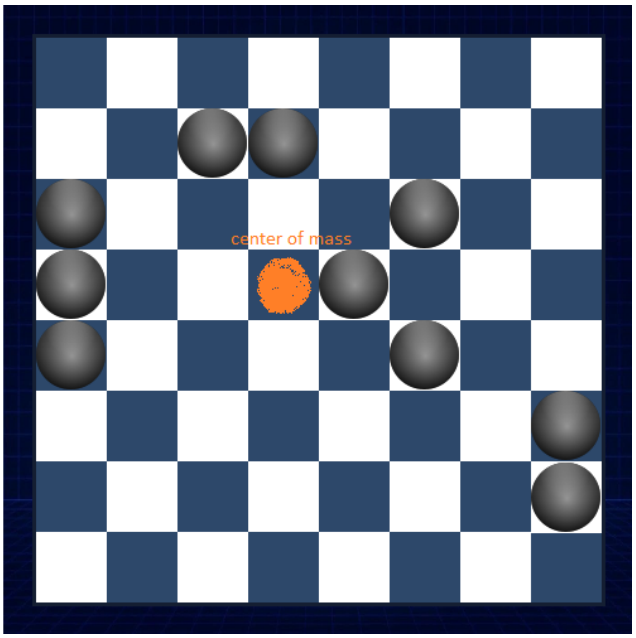


Figura 1.4.1. Centre de masses.

Mètode distancies_from_center

Donat el centre de masses, calcula la distància entre aquest punt i totes les fitxes. Es resta 10 punts per cada casella de distància amb el centre. Fem servir el mètode distance que està explicat a l'apartat *1.2. Heurística_2.*

```
public static int distances_from_center(Point center, ArrayList<Point> fitxes) {
	int score = 0;
	for(Point p: fitxes) {
		score += distance(center, p) * (-10);
	}
	return score;
}
```

Com es pot observar en la *Figura 1.4.2.* la forma de puntuar s'assembla a l'Heurística_3. Amb aquest tauler, la puntuació obtinguda sería de -210.

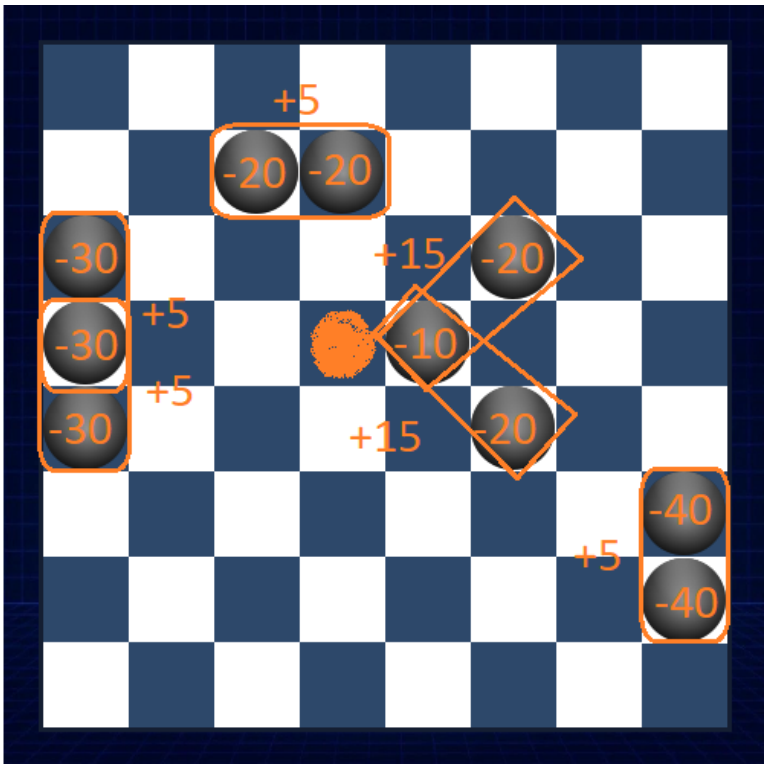


Figura 1.4.2. Tauler d'exemple.

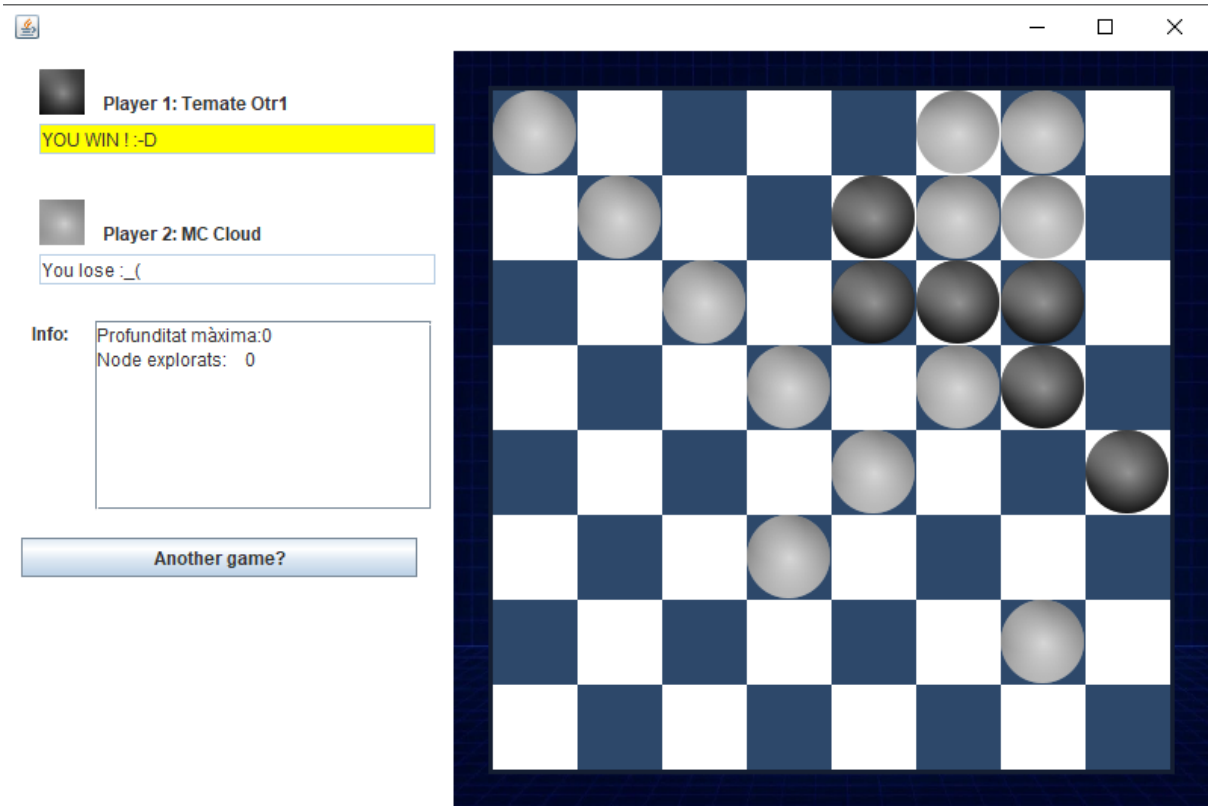


Figura 1.4.3. *Victoria de l'heuristica_4 contra MC Cloud.*

2. Millores de minmax

IDS

Per poder competir amb un temps limitat per baixar els nivells dels arbres, cal una millorar el nostre minimax perquè faci una cerca on partim d'un nivell de profunditat que coneixem que es asequible en un període de temps inferior al temps disponible.

Un cop acaba, ens guardem el resultat, incrementem la profunditat màxima i tornem a jugar, repetint el procés.

Un cop ens quedem sense temps, descartem la cerca actual que ha quedat a mitges i tornem la cerca anterior com a resultat.

Per fer això, tenim múltiples opcions:
Una seria ficar moltes condicions que vagin mirant si una variable *timeout* ha canviat de valor, indicant que hem arribat al timeout i fer un break a on es pugui durant l'execució.

Una altra alternativa, i la que hem aplicat, es la de fer ús de Interrupcions o Threads per parar l'execució del minmax. El funcionament es simple, el programa principal arranca una instancia que s'encarrega d'executar el minmax i després es queda esperant fins que un booleà *timeout* es fica a true, llavors es guarda l'últim resultat que ha obtingut el minmax i atura el fil perquè no continuï la seva execució.

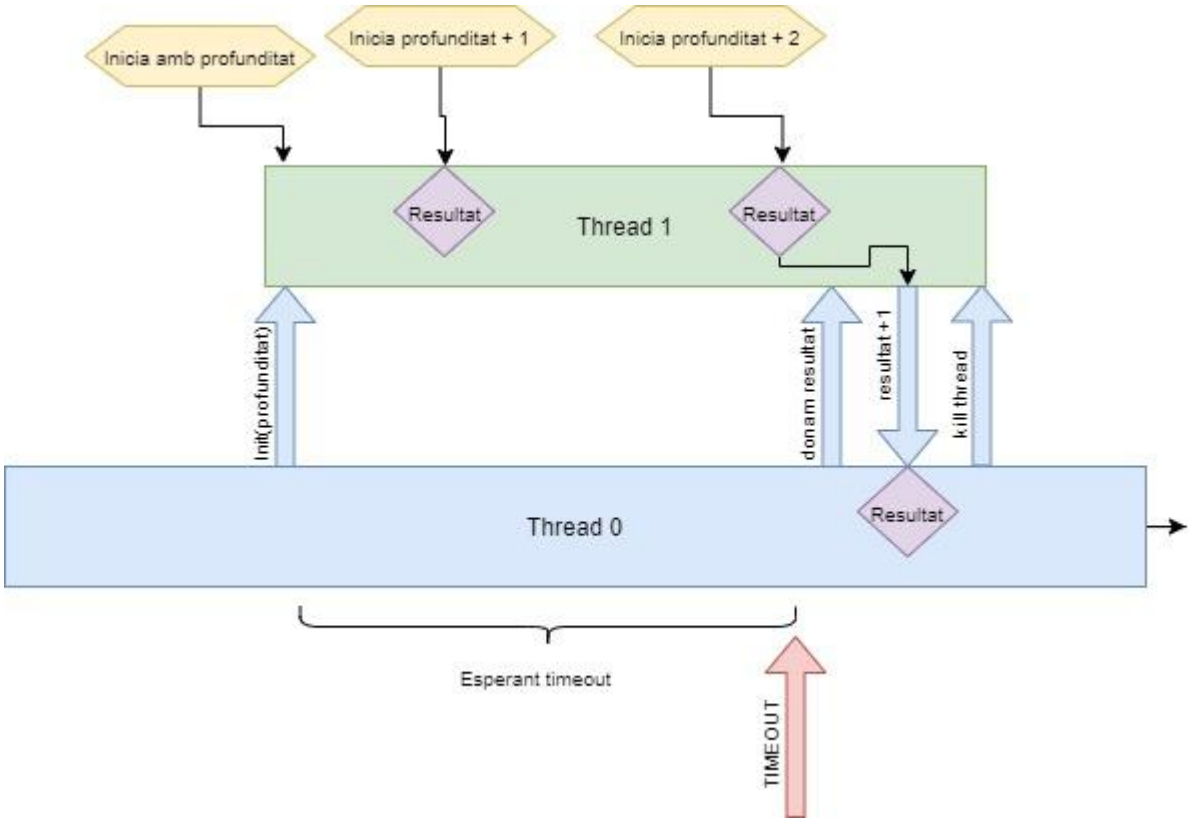


Figura 2.1. Diagrama de funcionament del minmax IDS

Zobrist

El següent pas logic es la implementació de zobrist, que ens permet tallar encara més nodes i tallar cicles. Lamentablement el temps no ens ha deixat acabar de implementar totes les millores que ens pot oferir, pero si algunes.

Primer hem de preparar una matriu 8x8x2 que identifica cada una de les posicions del tauler i si es una peca blanca o negra. Aquesta matriu s'inicialitzara un cop en tota la partida i tindrà numeros pseudo aleatoris.

Per identificar cada un dels estats, li asignem un numero hash calculat a partir del XOR de cada una de les fitxes amb la seva posició relativa en la matriu aleatoria anterior. La propietat del XOR ens permet calcular estats sense tenir que tornar a calcular totes les posicions de nou, és a dir, podem calcular parcialment nomes les modificacions del tauler.

Ens guardem cada un dels millors moviments de l'estat anterior i els fem servir per començar a mirar a partir de l'últim millor moviment, fent que la poda talli molts més camins.

3. Influencia de la poda alfa-beta

Per fer aquest estudi de l'influència de la poda alfa-beta, hem simulat la mateixa partida per que el resultat sigui el mateix en quant a estats del joc. Ens hem enfrontat contra el jugador `Bucky` utilitzant la nostra `heuristica2`.

Podem observar que gràcies a l'alfa-beta reduim 10 vegades el número de nodes explorats. Aquesta millora és molt gran i molt beneficiosa.

Algoritme min-max	Algoritme min-max alfa-beta
Nodes explorats totals: 20.009.528	Nodes explorats totals: 2.083.871

IDS		
Temps (segons)	**Nodes explorats**	**Profunditat máxima**
5	4.747.218	5
10	5.169.488	5
30	10.157.402	5
120	36.245.523	6

Zobrist (Incomplert)		
Temps (segons)	**Nodes explorats**	**Profunditat máxima**
5	2.673.906	5
10	5.169.488	5
30	10.157.402	5

4. Implicació

A continuació anem a tractar de classificar l'implicació de cada membre de l'equip en les diferents parts del projecte. En general hem dedicat el mateix temps de treball al projecte pero ens hem dividit algunes tasques per poder avançar més ràpidament, encara que quan teníem dubtes, sempre hem estat ajudant-nos i resolvent els problemes de forma conjunta.

En aquesta taula podem fer una aproximació de la repartició de tasques:

Tasca	Edu	Oriol
Documentació	70%	30%
Disseny i implementació de heuristiques	70%	30%
Timeout	30%	70%
Zobrist	40%	60%
Probar enfrontaments	50%	50%
Ús de git	50%	50%

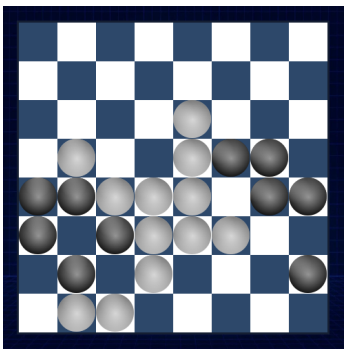
5. Enfrontaments versus IA MC Cloud

Per aquests enfrontaments farem servir el min-max alfa-beta IDS amb timeout de 10 segons.

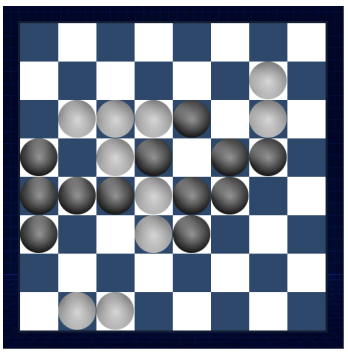
	10 Partides
Heuristica1	5 victòries - 5 derrotes
Heuristica2	9 victòries - 1 derrotes
Heuristica3	8 victòries - 2 derrotes
Heuristica4	4 victòries - 6 derrotes

Versus les nostres heurístiques

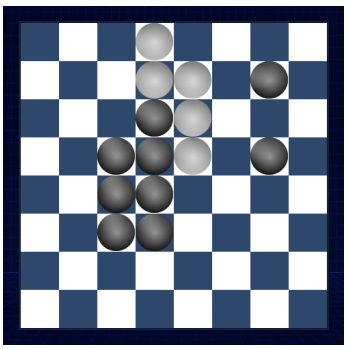
vs	Heuristica1	Heuristica2	Heuristica3	Heuristica4
Heuristica1	-	Heuristica2	Heuristica1	Heuristica4
Heuristica2	Heuristica2	-	Heuristica2	Heuristica2
Heuristica3	Heuristica1	Heuristica2	-	Heuristica4
Heuristica4	Heuristica1	Heuristica2	Heuristica4	-



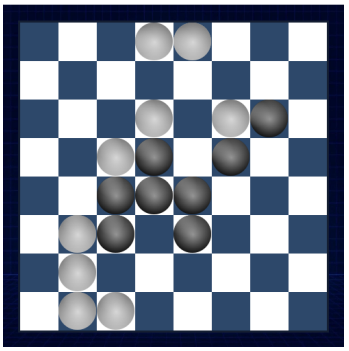
Heuristica1 vs Heuristica2
Guanyador: **Heuristica2**



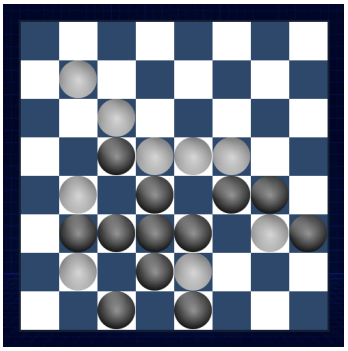
Heuristica1 vs Heuristica3
Guanyador: **Heuristica1**



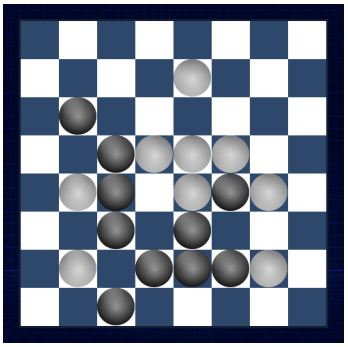
Heuristica1 vs Heuristica4
Guanyador: **Heuristica4**



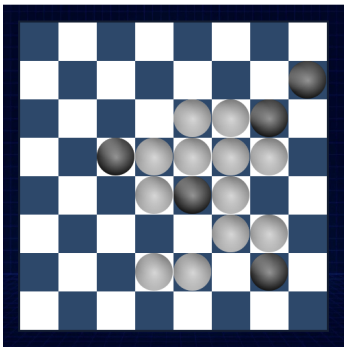
Heuristica2 vs Heuristica1
Guanyador: **Heuristica2**



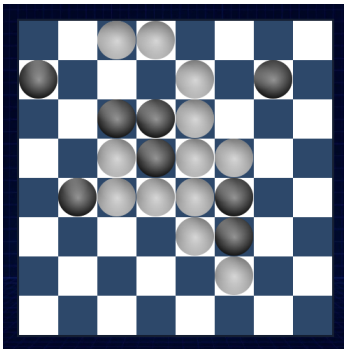
Heuristica2 vs Heuristica3
Guanyador: **Heuristica2**



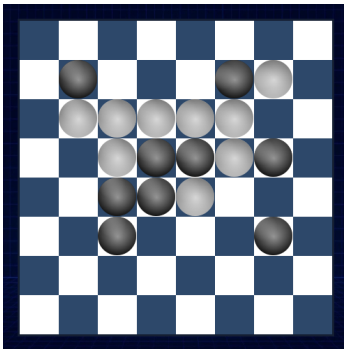
Heuristica2 vs Heuristica4
Guanyador: **Heuristica2**



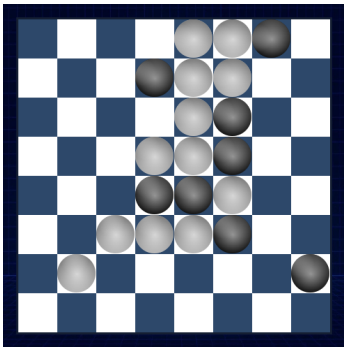
Heuristica3 vs Heuristica1
Guanyador: **Heuristica1**



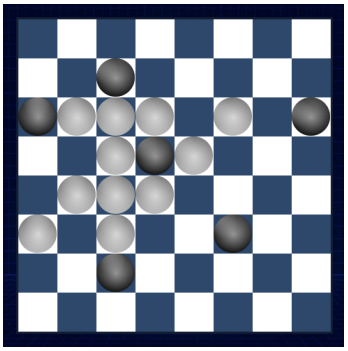
Heuristica3 vs Heuristica2
Guanyador: **Heuristica2**



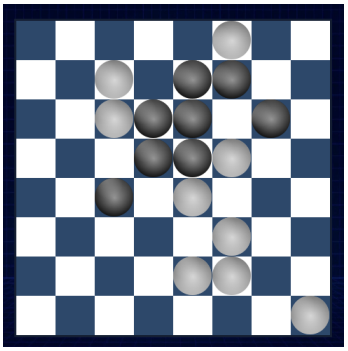
Heuristica3 vs Heuristica4
Guanyador: **Heuristica4**



Heuristica4 vs Heuristica1
Guanyador: **Heuristica1**



Heuristica4 vs Heuristica2
Guanyador: **Heuristica2**



Heuristica4 vs Heuristica3
Guanyador: **Heuristica4**

6. Git

Hem fet ús de git com a sistema de control de versions per ajudar-nos a mantenir el codi accesible i organitzat. Per accedir al projecte pots fer clic aquí.

7. Conclusions

Després de totes les proves realitzades, hem determinat que el millor rendiment l'obtenim fent ús de l'heurística 2 (*HEURISTICA_2*) i del minimax IDS (*MINMAX_IDS*).