DOCUMENTACIÓ CHECKERS!

Mickey La Rata

Ernest Anguera Aixalà, Naïm Barba Morilla

Taula de Continguts

EXPLICACIO DEL NOSTRE MINIMAX

Ens hem implementat un jugador automàtic utilitzant l'algorisme MiniMax per al joc de dames. En el nostre cas, aquest jugador es diu "MickeyLaRata". Aquest algorisme busca la millor jugada possible mitjançant una heurística específica, i limita la cerca fins a una determinada profunditat a l'arbre de possibilitats per optimitzar el rendiment.

El jugador automàtic comença analitzant l'estat actual del joc i determina quina és la millor jugada per al jugador actual. Primer, es defineix la profunditat màxima d'exploració de l'arbre de cerca, que pot especificar-se al constructor de la classe. A continuació, es genera una llista de possibles moviments disponibles per al jugador en el tauler.

L'algorisme utilitza una implementació del MiniMax que consisteix en una funció principal anomenada `miniMax`. Aquesta funció genera moviments possibles i evalua cada opció fins a la profunditat especificada, utilitzant les funcions auxiliars `maxValor` i `minValor`. S'utilitza també la tècnica d'alpha-beta pruning per reduir la quantitat de nodes explorats i millorar l'eficiència de l'algorisme.

L'avaluació de l'estat actual del joc es realitza mitjançant la funció `evaluarEstat`, que considera diversos factors, com la quantitat de peces, la posició al tauler, la proximitat als oponents i altres aspectes estratègics. Es té en compte la presència de reines, les peces a la fila de darrera, la proximitat als enemics i altres factors per determinar la millor heurística possible.

Finalment, el jugador automàtic selecciona la millor jugada basant-se en les evaluacions realitzades i retorna el moviment escollit. La implementació inclou també la gestió del temps d'execució per assegurar-se que el jugador automàtic no supera el límit de temps establert pel joc.

EXPLICACIO DEL NOSTRE MINIMAX-ID

En aquesta implementació, el nostre algorisme MiniMax Iterative Deepening Search forma la base de la estratègia d'un jugador automàtic per a les dames. Aquest algorisme s'enfoca en buscar la millor jugada possible, considerant una heurística específica, i limitant la cerca fins a una determinada profunditat de l'arbre de possibilitats. És important destacar que utilitzem la tècnica d'Iterative Deepening per garantir una millor gestió del temps d'execució.

INICIALITZACIO I ZOBRIST HASHING

Al començament, inicialitzem els atributs necessaris, com ara la profunditat màxima, el comptador de nodes explorats i altres variables relacionades amb la cerca. A més, implementem Zobrist Hashing amb l'ús d'una taula hash per emmagatzemar els estadis del joc ja explorats, optimitzant així la cerca en estats repetits.

ITERATIVE DEEPENING SEARCH

La cerca es realitza mitjançant Iterative Deepening, que implica generar moviments fins a una certa profunditat i emmagatzemar el millor moviment. Aquest procés es repeteix fins a l'arribada d'un possible timeout, moment en què es finalitza l'execució, retornant el millor moviment obtingut fins aleshores.

MINIMAX I ALPHA-BETA PRUNING

L'algorisme MiniMax s'implementa per determinar la millor jugada possible en una determinada profunditat. Es fa ús d'Alpha-Beta Pruning per optimitzar la cerca i reduir el nombre de nodes explorats. Això permet una millor eficiència en termes de temps d'execució.

HEURISTICA I AVALUACIO DE L'ESTAT DEL JOC

L'heurística específica és fonamental per avaluar els estats del joc. Considerem factors com la quantitat de peces, la proximitat als oponents, la seguretat de les peces i altres elements estratègics. Aquesta heurística influeix directament en la presa de decisions del jugador automàtic.

OPTIMITZACIONS I CONCLUSIO

L'ús de tècniques com Zobrist Hashing, Alpha-Beta Pruning i Iterative Deepening converteixen aquesta implementació de MiniMax en una estratègia potent per a les dames. Amb l'avaluació heurística, el jugador automàtic pot prendre decisions informades basades en la profunditat de cerca desitjada i la situació actual del joc. Tot plegat, forma un conjunt coherent que busca assegurar un rendiment òptim en termes d'eficiència i precisió estratègica.

HEURISTICA IMPLEMENTADA

Hem desenvolupant una heurística per avaluar l'estat actual d'un joc, amb un enfocament específic en un joc que sembla ser una variant del joc de dames. Anem a desglossar les diverses parts de l'heurística que hem implementat junts:

INICIALITZACIO I COMPTADOR DE NODES EXPLORATS:

Al principi de la funció evaluarEstat, incrementem el comptador de nodes explorats. Això és comú en algorismes de cerca, com ara els algorismes minimax, on avaluar diferents estats del joc implica explorar-los.

DIVISIO DE LA FUNCIO:

La funció evaluarEstat crida una funció auxiliar evaluarEstatAux per a cada jugador (jugador màxim i jugador mínim). Això ens permet avaluar l'estat del joc des de la perspectiva de cada jugador, tenint en compte les seves peces i la situació del taulell.

CALCUL DE DIVERSOS FACTORS:

Dins evaluarEstatAux, calculem diverses variables que representen diferents aspectes de l'estat del joc, com ara la quantitat de peces, la presència de reines, la distribució de les peces al taulell i altres factors que poden influir en la situació del joc.

AJUST DE LA HEURISTICA SEGONS LES NECESSITATS:

A continuació, ajustem la heurística segons algunes condicions. Per exemple, si la quantitat de peces de l'oponent és inferior o igual a 5, apliquem un bonus basat en la proximitat a les peces enemigues. Això pot indicar una estratègia específica en funció de la situació del joc.

ATRIBUCIO DE PUNTS I BONUS:

Sumem o restem punts a la heurística basant-nos en diversos factors, com ara la quantitat de peces regulars i reines, la ubicació de les peces al taulell, la seguretat o vulnerabilitat de les peces, i altres característiques específiques del joc.

RETORN DE LA HEURISTICA:

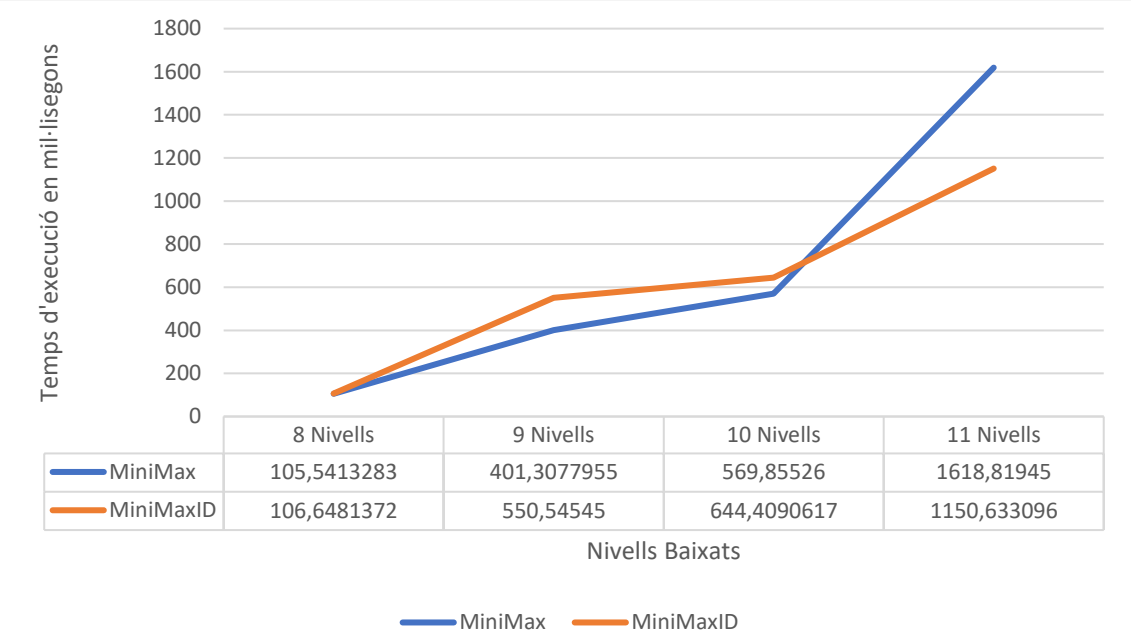
La funció finalment retorna el valor de la heurística, que serà utilitzat per l'algorisme global per prendre decisions sobre quins moviments realitzar al joc.

PARAMETRES AJUSTABLES:

Al llarg del codi, es destaca que els diferents "bonus" tenen comentaris indicant que es poden ajustar segons les necessitats o preferències específiques de l'usuari, proporcionant una flexibilitat per adaptar l'algorisme a diferents situacions de joc.

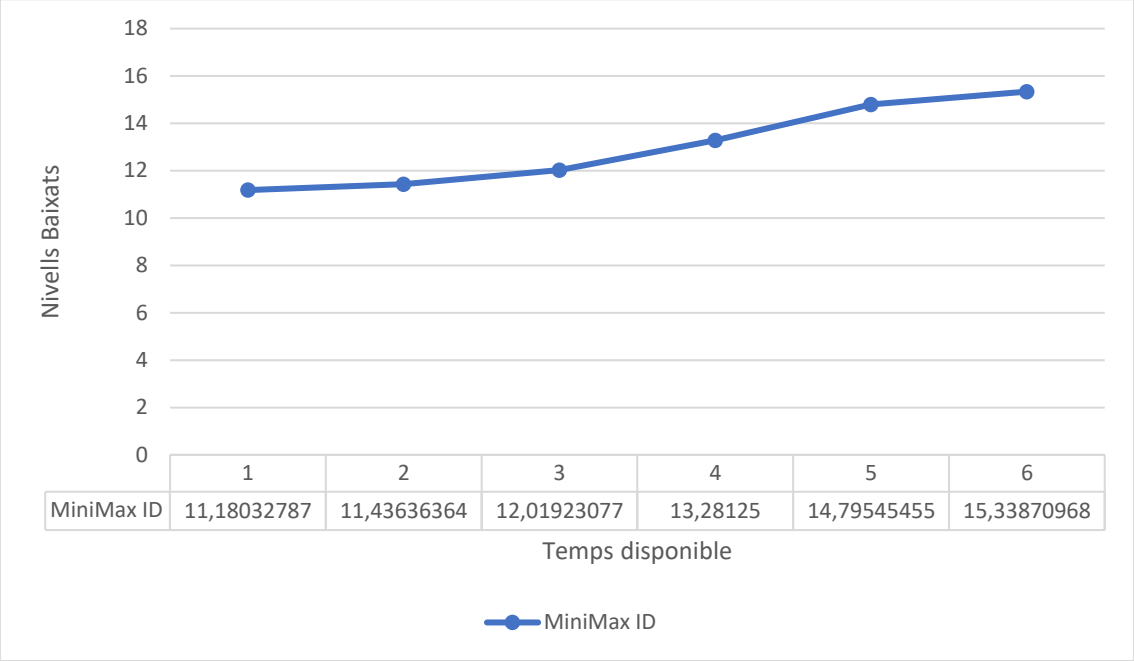
En resum, aquesta heurística està dissenyada per proporcionar una puntuació que reflecteixi la qualitat relativa de l'estat actual del joc per a un jugador determinat. Cada factor considerat i cada ajustament de bonificació contribueix a aquesta puntuació final, influenciant les decisions de l'algorisme en la presa de decisions durant la cerca.

COMPARACIO DEL TEMPS D'EXECUCIO



La representació gràfica revela que a mesura que es despleguen més nivells, l'eficàcia de l'agorisme MiniMax disminueix. Aquest fenomen s'explica per les millores implementades en l'algorisme MiniMax ID. A aquesta versió, no només s'aplica la poda alfa-beta i s'ordenen els moviments de manera decreixent (de més a menys importants), sinó que també es fa servir el Zobrist Hashing per emmagatzemar el millor moviment i aconseguir així una poda més eficient, amb una notable millora en el temps de processament.

NIVELLS BAIXATS SEGONS TEMPS AMB MINIMAXID



Observant la gràfica, es fa evident que a mida que permetem que el programa s'executi durant més temps, l'algoritme MiniMax ID explora més nivells. No obstant això, si ens fixem amb deteniment, notarem que el canvi més significatiu es produeix en l'interval [3-5]. En aquest punt, l'exploració augmenta a més d'un nivell per segon amb el temps disponible.

ESTRATEGIES D'OPTIMITZACIO UTILITZADES

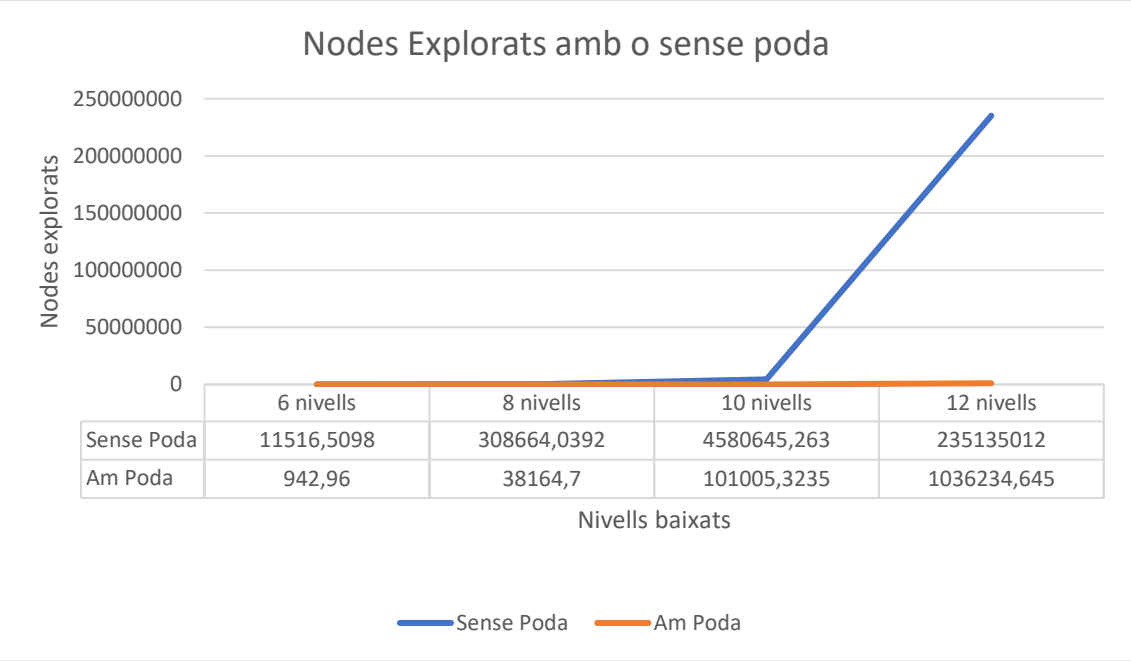
PODA ALPHA-BETA

En la nostra implementació del jugador automàtic, vam optar per utilitzar l'algoritme MiniMax amb poda alpha-beta per determinar la millor jugada possible en el joc de dames. Aquest algoritme ens permet explorar eficientment l'espai de possibles moviments fins a una profunditat especificada de l'arbre de cerca.

Quan rebem l'estat actual del joc a través de la funció **move(GameStatus s)**, inicialitzem les variables necessàries i cridem la funció principal **miniMax(GameStatus s)**. En aquesta funció, generem tots els possibles moviments a partir de l'estat actual del tauler i avaluem la heurística de cada moviment utilitzant les funcions `maxValor` i `minValor`, que representen els nodes MAX i MIN de l'arbre de cerca.

La poda alpha-beta es converteix en una part crucial de les funcions **minValor** i **maxValor**, ja que ens permet evitar explorar nodes que no milloraran els millors valors coneguts fins a aquest punt. Amb aquesta tècnica d'optimització, podem reduir significativament el nombre de nodes explorats, millorant així el rendiment del nostre jugador automàtic.

Per a comprovar l'efectivitat de la poda alpha-beta, hem fet diferents execucions del programa amb i sense poda:



Com es pot observar en el gràfic, la realització de la poda optimitza significativament l'execució de l'algoritme. Aquesta optimització possibilita descendir molts més nivells en el mateix temps d'execució, a més de reduir el temps d'execució per a un nombre específic de nivells.

En resum, la nostra implementació utilitza l'algoritme MiniMax amb poda alpha-beta per prendre decisions informades i eficients en cada moviment del joc de dames. Aquest enfocament ens permet trobar la millor estratègia tenint en compte una heurística específica, tot mantenint un rendiment raonable en termes de temps d'execució.

ZOBRIST HASHING

El Zobrist Hashing és una tècnica utilitzada per generar de manera eficient valors hash per a posicions de tauler en jocs de tauler. En el context del teu programa, el Zobrist Hashing es implementa a la classe `ProfeNoFuncaStatus` per calcular un valor hash únic per a cada estat del joc.

Funcionament i Implementació del Zobrist Hashing a ProfeNoFuncaStatus:

Array de Hash Zobrist (zobrist):

Un array tridimensional zobrist és creat per emmagatzemar els valors hash Zobrist per a diferents tipus de cel·les i les seves posicions al tauler de joc.

```
1. private static int[][][] zobrist;
```

En el bloc estàtic, aquests valors es inicialitzen amb nombres aleatoris:

```
 1.    static {
 2.        zobrist = new int[8][8][4];
 3.        Random rand = new Random();
 4.        for (int i = 0; i < 8; i++) {
 5.            for (int j = 0; j < 8; j++) {
 6.                for (int k = 0; k < 4; k++) {
 7.                    zobrist[i][j][k] = rand.nextInt();
 8.                }
 9.            }
10.        }
11.        black_to_move = rand.nextInt();
12.    }
13.
```

Variables Addicionals:

- *black_to_move:* Un valor aleatori que representa el jugador actual que realitza el moviment.
- *hash:* La variable que emmagatzemarà el valor hash actual.
- *hash_updated:* Una bandera per indicar si el valor hash ha estat actualitzat des de l'última vegada que es va moure una peça.

```
1.    private static int black_to_move;
2.    private int hash;
3.    private boolean hash_updated = false;
```

Mètode hashCode():

Aquest mètode s'encarrega de calcular el valor hash basat en la posició actual del tauler i la informació del jugador actual.

```
1.    @Override
2.    public int hashCode() {
3.        if (hash_updated) {
4.            return hash;
5.        }
6.        hash = 0;
7.        if (this.getCurrentPlayer() == PlayerType.PLAYER1) {
8.            hash ^= black_to_move;
9.        }
10.       for (int i = 0; i < this.getSize(); ++i) {
11.           for (int j = 0; j < this.getSize(); ++j) {
12.               CellType casella = this.getPos(i, j);
13.               switch (casella) {
14.                   case P1 ->
15.                       hash ^= zobrist[i][j][0];
16.                   case P2 ->
17.                       hash ^= zobrist[i][j][1];
18.                   case P1Q ->
19.                       hash ^= zobrist[i][j][2];
20.                   case P2Q ->
21.                       hash ^= zobrist[i][j][3];
22.                   default -> {
23.                       // No cal fer res per a les cel·les buides
24.                   }
25.               }
26.           }
27.       }
28.       hash_updated = true;
29.       return hash;
30.   }
```

L'operació XOR (`^`) s'utilitza per combinar els valors hash de diferents cel·les al tauler.

Mètode equals(Object obj):

Es sobrescriu el mètode `equals` per comparar els valors hash de dos estats de joc i determinar si són iguals.

```
1.    @Override
2.    public boolean equals(Object obj) {
3.        if (this == obj) {
4.            return true;
5.        }
6.        if (obj == null) {
7.            return false;
8.        }
9.        if (getClass() != obj.getClass()) {
10.           return false;
11.       }
12.       final ProfeNoFuncaStatus other = (ProfeNoFuncaStatus) obj;
13.       return this.hashCode() == other.hashCode();
14.   }
```

Mètode movePiece(List<Point> list):

Aquest mètode es sobrescriu per actualitzar la bandera `hash_updated` després de moure una peça al tauler.

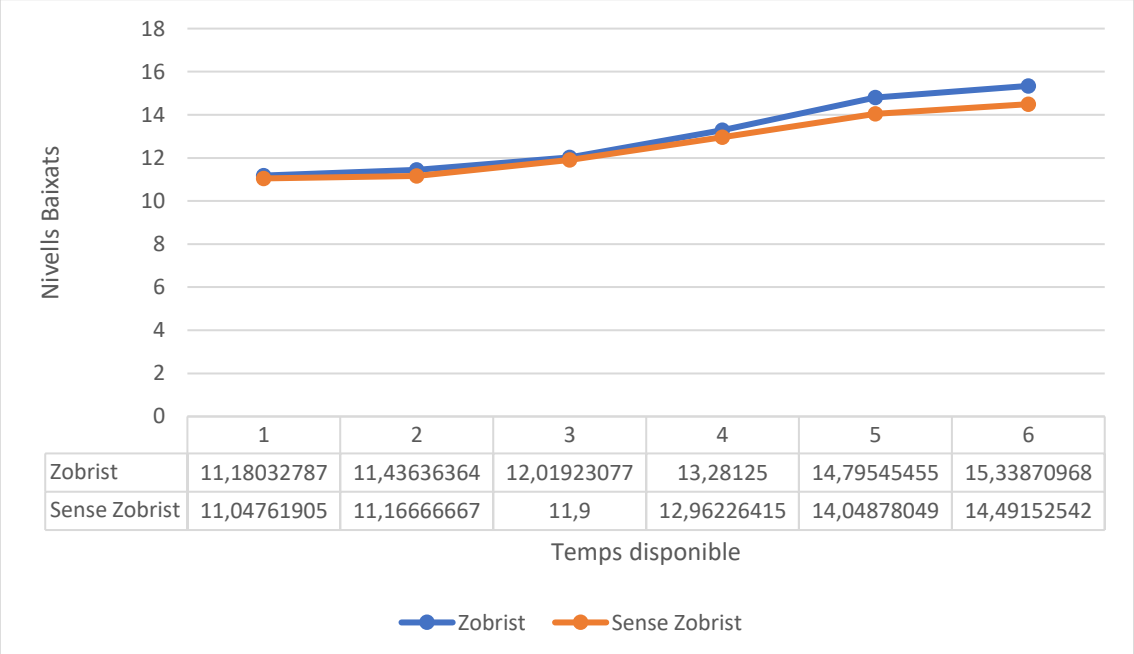
```
1.    @Override
2.    public void movePiece(List<Point> list) {
3.        super.movePiece(list);
4.        hash_updated = false;
5.    }
```

Ús del Zobrist Hashing a la Classe PlayerID1:

Dins de la classe `PlayerID1`, s'utilitza un HashMap per desar informació sobre l'estat del joc i els millors moviments ja explorats. Aquest HashMap fa servir valors hash generats amb Zobrist Hashing com a clau i conté un objecte de la classe `GameInfo` com a valor. Aquest objecte inclou dades sobre el moviment realitzat, el nombre de nivells per sota, el valor heurístic i el tipus de jugador.

La funció principal de la classe `PlayerID1` és gestionar aquest HashMap per prendre decisions eficaces durant el joc i explorar l'espai d'estats de manera optimitzada. Els valors hash ajuden a identificar ràpidament si l'estat actual del joc ja ha estat explorat. En cas afirmatiu, es fa servir la informació emmagatzemada per prendre decisions més informades. Aquesta tècnica optimitza l'algorisme de presa de decisions i evita explorar repetidament els mateixos estats del joc

En el codi, el HashMap s'utilitza per recuperar el millor moviment guardat per a un estat específic del joc, si existeix. Quan aquest moviment es troba, es col·loca a la primera posició de la llista de camins, augmentant així l'eficiència de la poda i permetent baixar més nivells en cada execució del programa. Per demostrar-ho, hem creat una gràfica que mostra el nombre de nivells baixats segons el temps disponible, amb i sense l'ús de Zobrist hashing:



	1	2	3	4	5	6
Zobrist	11,18032787	11,43636364	12,01923077	13,28125	14,79545455	15,33870968
Sense Zobrist	11,04761905	11,16666667	11,9	12,96226415	14,04878049	14,49152542

Com es pot notar, a mesura que es disposa de més temps disponible, l'ús de Zobrist es torna més òptim. Aquest fenomen es deu al fet que amb més temps, es visiten un major nombre d'estats del joc, augmentant així la utilitat de la tècnica de poda. En altres paraules, la capacitat de reduir el nombre d'estats explorats proporcionada per Zobrist Hashing esdevé més beneficiosa quan hi ha una extensió temporal més gran disponible per explorar les diverses possibilitats del joc.

REPARTICIO DE LES TASQUES

Per a aquest treball, ens hem repartit les tasques de manera bastant equitativa, a la següent taula es pot veure el repartiment de tasques:

PlayerMiniMax	Ernest	50%
	Naïm	50%
PlayerID	Ernest	60%
	Naïm	40%
Heurística	Ernest	40%
	Naïm	60%
Documentació	Ernest	50%
	Naïm	50%

ENLLAÇ DEL REPOSITORI DE GITHUB

https://github.com/eur1p3des/Checkers.git